VISIT…

Wrox Programmer to Programmer™

Professional

Apache Tomcat 6

Vivek Chopra, Sing Li, Jeff Genender

wrox

Updates, source code, and Wrox technical support at www.wrox.com

Professional
Apache Tomcat 6

Vivek Chopra
Sing Li
Jeff Genender

Wiley Publishing, Inc.

Professional
Apache Tomcat 6

Introduction	xxiii
Chapter 1: Apache Tomcat	1
Chapter 2: Web Applications: Servlets, JSPs, and More	13
Chapter 3: Tomcat Installation	29
Chapter 4: Tomcat Architecture	51
Chapter 5: Basic Tomcat Configuration	69
Chapter 6: Advanced Tomcat Features	103
Chapter 7: Web Application Configuration	135
Chapter 8: Web Application Administration	173
Chapter 9: Class Loaders	205
Chapter 10: HTTP Connectors	221
Chapter 11: Tomcat and Apache HTTP Server	243
Chapter 12: Tomcat and IIS	285
Chapter 13: JDBC Connectivity	309
Chapter 14: Tomcat Security	335
Chapter 15: Shared Tomcat Hosting	387
Chapter 16: Monitoring and Managing Tomcat with JMX	419
Chapter 17: Clustering	455
Chapter 18: Embedded Tomcat	493
Chapter 19: Logging	505
Chapter 20: Performance Testing	533
Chapter 21: Performance Tuning	561
Appendix A: Tomcat and IDEs	585
Appendix B: Apache Ant	597
Index	621

Professional
Apache Tomcat 6

Vivek Chopra
Sing Li
Jeff Genender

Wiley Publishing, Inc.

Professional Apache Tomcat 6

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2007 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-471-75361-2

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data

Chopra, Vivek.
Professional Apache Tomcat 6 / Vivek Chopra, Sing Li, and Jeff Genender.
p. cm.
Includes index.
ISBN 978-0-471-75361-2 (paper/website)
1. Apache Tomcat. 2. Web servers. 3. Web site development. 4. Internet programming. I. Li, Sing. II. Genender, Jeff M. III. Title.
TK5105.8885.A63C47 2007
005.7'1376—dc22

2007020134

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at http://www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Website is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Website may provide or recommendations it may make. Further, readers should be aware that Internet Websites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

To Rebecca and Rohan, thanks for all your patience and support.
—Vivek

To my guiding light and spiritual support for the last two decades, Kim.
—Sing

To my wonderful wife, Nazarena, and my children, Madisyn, Weston, and Coleton.
I could not have done this without you.
—Jeff

About the Authors

Vivek Chopra has more than 13 years of experience as a software architect, developer, and team lead and has worked in a number of Silicon Valley companies and startups. He writes actively on technology and is the author of more than half a dozen books on Java, open source software, XML, and Web services. Vivek has pending patents on Web service technologies, and has been a Java Community Process (JCP) member for the past three years. He also serves on the expert group for JSR 280 (XML API for Java ME).

Sing Li (who was bitten by the microcomputer bug in the late 1970s) has grown up with the Microprocessor Age. His first personal computer was a $99 do-it-yourself Netronics COSMIC ELF with 256 bytes of memory, mail-ordered from the back pages of *Popular Electronics* magazine. A 20-year industry veteran, Sing is a system developer, open source software contributor, and freelance writer specializing in Java technology, and embedded and distributed systems architecture. He regularly writes for several popular technical journals and e-zines, and is the creator of the "Internet Global Phone," one of the very first Internet phones available. He has authored and co-authored a number of books across diverse technical disciplines including Geronimo, Tomcat, JSP, servlets, XML, Jini, media streaming, device drivers, and JXTA.

Jeff Genender has over 18 years of software architecture, team lead, and development experience in multiple industries. Jeff is an active committer and Project Management Committee (PMC) member for Apache Geronimo, and a committer on OpenTerracotta, OpenEJB, ServiceMix, and Mojo (Maven plugins). Jeff also serves as a member of the Java Community Process (JCP) expert group for JSR-313 (Java Platform, Enterprise Edition 6 [Java EE 6] Specification) as a representative of the Apache Software Foundation. Jeff is an open source evangelist and has successfully brought open source development efforts, initiatives, and success stories into a number of Global 2000 companies, saving these organizations millions in licensing costs.

Credits

Executive Editor
Bob Elliott

Development Editor
Sydney Jones

Technical Editors
Rupert Jones
Anne Horton

Copy Editor
Nancy Rapoport

Editorial Manager
Mary Beth Wakefield

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Joseph B. Wikert

Project Coordinator, Cover
Adrienne Martinez

Proofreader
Kathryn Duggan

Indexer
Johnna VanHoose Dinse

Anniversary Logo Design
Richard Pacifico

Contents

Introduction xxiii

Chapter 1: Apache Tomcat 1

Humble Beginnings: The Apache Project 2
The Apache Software Foundation 3
Tomcat 3
Distributing Tomcat: The Apache License 4
Comparison with Other Licenses 5
The Big Picture: Java EE 6
Java APIs 6
The Java EE APIs 7
Java EE Application Servers 8
"Agree on Standards, Compete on Implementation" 8
Tomcat and Application Servers 9
Tomcat and Web Servers 9
Summary 10

Chapter 2: Web Applications: Servlets, JSPs, and More 13

A Brief History of Web Applications 13
CGI Scripts: The First Mechanism for Dynamic Content 13
Server Side Java: Servlets 14
JavaServer Pages 19
JSP Tag Libraries 22
JSP EL 23
MVC Architecture 24
Using Appropriate Web Technologies 25
Building and Distributing Web Applications 26
Summary 27

Chapter 3: Tomcat Installation 29

Installing the Java Virtual Machine 29
Installing the JVM on Windows 30
Installing the JVM on Linux 32

Installing Tomcat **34**
Deciding Which Distribution to Install 34
Verifying the Downloaded File 35
Tomcat Windows Installer 36
Installing Tomcat on Windows Using the ZIP File 41
Installing Tomcat on Linux 42
Building Tomcat from Source **44**
Do You Need to Build Tomcat from the Source Code? 44
Downloading the Source Release 44
Subversion Repository 45
Building a Source Release 45
The Tomcat Installation Directory **46**
Installing APR **47**
Troubleshooting and Tips **48**
Class Version Error 49
The Port Number Is in Use 49
Running Multiple Instances 49
A Proxy Is Blocking Access 49
Summary **50**

Chapter 4: Tomcat Architecture **51**

Tomcat Directory Overview **51**
bin Directory 52
conf Directory 52
lib Directory 53
logs Directory 53
temp Directory 53
webapps Directory 53
work Directory 54
An Overview of Tomcat Architecture **54**
The Server 55
The Service 56
The Remaining Classes in the Tomcat Architecture 59
Connector Architecture **59**
Communication Paths 60
Connector Protocols 61
Choosing a Connector 63
Lifecycle **64**
Lifecycle Interface 65
LifecycleListener Interface 65
Configuration by Architecture **66**
Summary **67**

Chapter 5: Basic Tomcat Configuration 69

Tomcat 6 Configuration Essentials 70
Files in $CATALINA_HOME/conf 71
Basic Server Configuration 71
Server Configuration via the Default server.xml 72
Operating Tomcat in Application Server Configuration 75
Web Application Context Definitions 82
The Default context.xml File 82
Authentication and the tomcat-users.xml File 86
The Default Deployment Descriptor — web.xml 86
How server.xml, Context Descriptors, and web.xml Work Together 91
Fine-Grained Access Control: catalina.policy 94
catalina.properties: Finer-Grained Control over Access Checks 97
Bootstrapping Configuration 97
A Final Word on Differentiating Between Configuration and Management 98
Tomcat 6 Web-Based GUI Configurator 98
Summary 100

Chapter 6: Advanced Tomcat Features 103

Valves — Interception Tomcat-Style 104
Standard Valves 104
Access Log Implementation 105
Scope of Log Files 106
Single Sign-On Implementation 108
Multiple Sign-On Without the Single Sign-On Valve 109
Configuring a Single Sign-On Valve 111
Form Authenticator Valve 112
Restricting Access via a Request Filter 112
Remote Address Filter 112
Remote Host Filter 113
Configuring Request Filter Valves 113
Request Dumper Valve 114
Persistent Sessions 115
The Need for Persistent Sessions 115
Configuring a Persistent Session Manager 115
JNDI Resource Configuration 118
What Is JNDI? 118
Tomcat and JNDI 119
Typical Tomcat JNDI Resources 120
Configuring Resources via JNDI 121

Configuring a JDBC DataSource 124
Configuring Mail Sessions 126
Configuring Lifecycle Listeners 129
Lifecycle Events Sent by Tomcat Components 129
The <Listener> Element 129
Tomcat 6 Lifecycle Listeners Configuration 130
Summary 133

Chapter 7: Web Application Configuration 135

Understanding the Contents of a Web Application 135
Public Resources 136
The WEB-INF Directory 138
The META-INF Directory 139
Understanding the Deployment Descriptor (web.xml) 140
The Servlet 2.3–Style Deployment Descriptor 141
The Servlet 2.4/2.5–Style Deployment Descriptor 154
Summary 171

Chapter 8: Web Application Administration 173

Sample Web Application 173
Tomcat Manager Application 175
Enabling Access to the Manager Application 176
Manager Application Configuration 178
Tomcat Manager: Web Interface 180
Displaying Tomcat Server Status 180
Managing Web Applications 181
Deploying a Web Application 182
Tomcat Manager: Managing Applications with Ant 182
Known Issue: Failure While Undeploying Web Applications on Windows 188
Tomcat Manager — Using HTTP Requests 189
List Deployed Applications 190
Deploying a New Application 190
Installing/Deploying Applications in Tomcat 6 191
Deploying a New Application Remotely 192
Deploying a New Application from a Local Path 192
Reloading an Existing Application 194
Listing Available JNDI Resources 195
Listing OS and JVM Properties 196
Stopping an Existing Application 196
Starting a Stopped Application 197
Undeploying a Web Application 198

Displaying Session Statistics 198
Querying Tomcat Internals Using the JMX Proxy Servlet 199
Setting Tomcat Internals Using the JMX Proxy Servlet 200
Possible Errors 200
Security Considerations 201
Tomcat Deployer 203
Summary 203

Chapter 9: Class Loaders 205

Class Loader Overview 205
Standard Java SE Class Loaders 207
More on Class Loader Behavior 210
Creating a Custom Class Loader 211
Why Is a Custom Class Loader Needed for Tomcat? 211
Security and Class Loaders 212
Class Loader Delegation 212
Core Class Restriction 212
Separate Class Loader Namespaces 213
SecurityManager 213
Tomcat and Class Loaders 214
System Class Loader 215
Endorsed Standards Override Mechanism 215
Common Class Loader 215
Web Application Class Loader 216
Dynamic Class Reloading 217
Common Class Loader Pitfalls 218
Packages Split Among Different Class Loaders 218
Singletons 218
XML Parsers 219
Summary 220

Chapter 10: HTTP Connectors 221

HTTP Connectors 222
Tomcat 6 HTTP/1.1 Connector 223
The Advanced NIO Connector 227
Comet Asynchronous IO Support 228
The Native APR Connector 228
Configuring Tomcat for CGI Support 232
Configuring Tomcat for SSI Support 234
Configuring the Tomcat 6 SSI Servlet 235
Configuring the Tomcat 6 SSI Filter 237

Running Tomcat Behind a Proxy Server **238**
Performance Tuning **239**
Tunable Configuration Attributes 239
TCP/IP Stack Tuning Tips 240
Front-Ending Tomcat 6 with a Web Server **241**
Summary **242**

Chapter 11: Tomcat and Apache HTTP Server **243**

The AJP Connector Architecture **244**
The Native Code Apache Modules 244
The Apache JServ Protocol 245
The AJP Connector 245
Apache Web Server Frontend or Tomcat Standalone **246**
Understanding Tomcat Workers **246**
Multiple Tomcat Workers 246
Configuring Apache Server to Work with Multiple Tomcat Workers — the workers.properties File 247
Connecting Tomcat with Apache **251**
Tomcat 6 Configuration 251
Apache Web Server Configuration 252
Using the mod_jk Module 253
Using the mod_proxy Module 259
Configuring SSL for Apache Web Server **263**
Configuring mod_ssl for Apache 264
Testing the SSL-Enabled Apache Setup 269
SSL-Enabled Apache-Tomcat Setup 271
Tomcat Load Balancing with Apache **273**
Changing CATALINA_HOME in the Tomcat Startup Files 274
Setting Different AJP Connector Ports 275
Setting Different Server Ports 275
Disabling the Default HTTP/1.1 Connector 276
Setting the jvmRoute in the Standalone Engine 276
Commenting Out the Catalina Engine 277
Directives in httpd.conf 277
Workers Configuration in workers.properties 277
Testing the Load Balancer **279**
Testing Sticky Sessions 280
Testing Round-Robin Behavior 281
Testing with Different Load Factors 283
Summary **284**

Chapter 12: Tomcat and IIS 285

Role of the ISAPI Plug-in 285
Connecting Tomcat with IIS 286
Verifying Tomcat and IIS Installations 287
Configuring the JK Connector 288
Installing the ISAPI Plug-in 288
Configuring Tomcat Workers 289
Configuring the Request Forwarding Rules 291
Optionally Configure URL Rewrite Rules 292
Updating the Windows Registry for the ISAPI Plug-in 292
IIS 5 Isolation Mode (IIS 6 Only) 295
Creating a Virtual Directory Under IIS 296
Adding the ISAPI Plug-in as an IIS Filter 300
Authorizing the ISAPI Plug-in as a Web Application Extension (IIS 6 Only) 302
Testing the Final Setup 303
Troubleshooting Tips 303
Using SSL 305
Scalable Architectures with IIS and Tomcat 305
Distributing Web and Application Server Deployments 306
Multiple Tomcat Workers 307
Load-Balanced AJP Workers 307
Summary 307

Chapter 13: JDBC Connectivity 309

JDBC Basics 310
Establishing and Terminating Connections to RDBMSs 311
Evolving JDBC Versions 311
JDBC Driver Types 312
Database Connection Pooling 313
A Problem with Connection Pooling 314
Tomcat and the JDBC Evolution 315
JNDI Emulation and Pooling in Tomcat 6 315
Preferred Configuration: JNDI Resources 317
The Resource Tag 317
Hands-On JNDI Resource Configuration 319
Testing the JNDI Resource Configuration 324
Alternative JDBC Configuration 326
Alternative Connection Pool Managers 326
About the c3p0 Pool Manager 326
Deploying the c3p0 Pooling Manager 327

Obtaining JDBC Connections Without JNDI Lookup 327
Testing Non-JNDI Pool Access with c3p0 329
Obtaining a Connection with JNDI Mapping 330
Testing c3p0 with Tomcat 6 JNDI-Compatible Lookup 331
Deploying Third-Party Pools 332
Summary 332

Chapter 14: Tomcat Security 335

Verifying Tomcat Download Integrity 336
Verifying the MD5 DIGEST 336
Using PGP to Verify the Download 338
Securing the Tomcat Server Installation 340
Removing Default Applications 341
ROOT and tomcat-docs 341
System Applications — manager and host-manager 341
Tying Down System Application Access Security 341
Removing JSP and Servlet Examples 342
Changing the SHUTDOWN Command 342
Running Tomcat with a Special Account 342
Creating a Non-Privileged Tomcat User 343
Running Tomcat with the Tomcat User 343
Securing the File System 344
Windows File System 344
Linux File System 346
Securing the Java Virtual Machine 346
Overview of the Security Manager 347
Using the Security Manager with Tomcat 350
Recommended Security Manager Practices 353
Securing Web Applications 355
Authentication and Realms 355
Security Realms 360
Encryption with SSL 377
JSSE 378
Protecting Resources with SSL 381
Securing DefaultServlet 383
Disabling Directory Listing 383
Disabling an Invoker Servlet, SSI, and CGI Gateway 384
Host Restriction 384
Summary 384

Chapter 15: Shared Tomcat Hosting **387**

Virtual Hosting Concepts **387**
Virtual Hosting in Apache **388**
Example Deployment Scenario 388
IP-Based Virtual Hosting in Apache 389
Name-Based Virtual Hosting in Apache 392
Virtual Hosting in Tomcat **395**
Example Deployment Scenario 396
Tomcat as a Standalone Server 398
Tomcat with Apache 405
Configuring Apache 406
The Tomcat Host-Manager Application **409**
Virtual Hosting Issues: Stability, Security, and Performance **409**
Tuning Virtual Hosting Settings in Tomcat **410**
Creating Separate JVMs for Each Virtual Host 410
Setting Memory Limits on the Tomcat JVM 414
Using Java Security Manager Restrictions 416
Summary **417**

Chapter 16: Monitoring and Managing Tomcat with JMX **419**

The Requirement to Be Manageable **420**
All About JMX **422**
The JMX Architecture 422
Instrumentation Level 424
Agent Level 425
Distributed Services Level 427
JMX Remote API **428**
An Anthology of MBeans **428**
Standard MBeans 428
Dynamic MBeans 428
Model MBeans 429
Open MBeans 429
JMX Manageable Elements in Tomcat 6 **429**
Manageable Tomcat 6 Architectural Components 430
Manageable Nested Components 430
Manageable Runtime Data Objects 430
Manageable Resource Object 436
Accessing Tomcat 6's JMX Support via the Manager Proxy **441**
Working with the JMX Proxy 442

Modifying MBean Attributes 444
Using jconsole GUI to Monitor Tomcat 447
Configuring Tomcat for Remote Monitoring 450
Summary 452

Chapter 17: Clustering 455

Clustering Benefits 456
Scalability and Clustering 456
The Need for High Availability 457
Clustering Basics 457
Master-Backup Topological Pattern 457
Fail-Over Behavioral Pattern 458
Tomcat 6 Clustering Model 459
Load Balancing 460
Session Sharing 461
Working with Tomcat 6 Clustering 465
Session Management in Tomcat 6 465
The Role of Cookies and Modern Browsers 466
Configuring a Tomcat 6 Cluster 466
Common Front End: Load Balancing via Apache mod_jk 471
Preparation for Using Different Session-Sharing Backends 472
Backend 1: In-Memory Replication Configuration 472
Backend 2: Persistent Session Manager with a Shared File Store 484
Backend 3: Persistent Session Manager with a JDBC Store 487
Testing a Tomcat Cluster with JDBC Persistent Session Manager Backend 490
The Complexity of Clustering 490
Clustering and Performance 490
Clustering and Response Time 491
Solving Performance Problems with Clustering 491
Summary 491

Chapter 18: Embedded Tomcat 493

Importance of Embedded Tomcat in Modern System Design 494
Typical Embedded Application Scenarios 495
Developing with Embedded Tomcat 495
Summary 503

Chapter 19: Logging **505**

Changes from Tomcat 5 **505**
log4j **506**
log4j Architecture 506
log4j Installation and Configuration 509
A Tutorial Introduction to log4j 514
More log4j Recipes 515
log4j Performance Tips 527
JULI **527**
Java Logging Architecture 527
A Tutorial Introduction to JULI 529
Log Files Analysis **531**
Summary **532**

Chapter 20: Performance Testing **533**

Performance Concepts **533**
What to Measure 533
Scalability and Performance 534
Understanding the User's Perspective 535
Measuring Performance 535
JMeter **537**
Installing and Running JMeter 537
Making and Understanding Test Plans with JMeter 538
JMeter Features 542
Distributed Load Testing 554
Interpreting Test Results 555
Alternatives to JMeter **558**
What to Do After Performance Testing **558**
Summary **559**

Chapter 21: Performance Tuning **561**

Performance Tuning Best Practices **561**
Step 1: Set Up a Test Bed 562
Step 2: Test Performance and Identify the Baseline 563
Step 3: Diagnose Performance Bottlenecks 564
Diagnosing Tomcat Performance Issues **564**
Tomcat Performance Tuning Tips **566**
Tuning the JVM Parameters 567
Precompiling JSPs 569

Tuning Tomcat Configuration 571
Using Web Servers for Static Content, When Appropriate 582
Summary 584

Appendix A: Tomcat and IDEs 585

Eclipse 585
Debugging a Remote Web Application in Eclipse 586
Deploying and Debugging Local Web Applications Using the Sysdeo Tomcat Plugin 589
Deploying and Debugging Web Applications Using the Web Tools Platform 591
Managing Web Application Deployment Using Apache Ant and Eclipse 593
NetBeans 593
Debugging a Remote Web Application In NetBeans 594
Debugging a Web Application Inside NetBeans 596
Summary 596

Appendix B: Apache Ant 597

Installing Ant 598
Introduction to Ant 598
More Command-Line Options 601
Ant Recipes 602
Building Web Applications with Ant 602
Compiling JSPs 608
Reusable Ant Scripts Using Property Files and Command-Line Parameters 609
Build Logs 610
Build Notifications via E-mail 611
Ant and Source Control Systems 613
Automated Testing 614
Continuous Integration 615
Ant Task Reference 615
Summary 619

Index 621

Introduction

Professional Apache Tomcat 6 is primarily targeted toward administrators and engineers responsible for Tomcat configuration, performance tuning, system security, or deployment architecture. This book doesn't cover Web application development using Tomcat. A lot of other books, such as our *Beginning JavaServer Pages* (Wrox Press, ISBN 0-7645-7485-X), fulfill this need. Instead, this book focuses on its primary audience — i.e., Tomcat administrators — and tries to provide what this audience needs as best as it can.

This is the third edition in our Apache Tomcat series. Our first edition, *Professional Apache Tomcat*, covered Tomcat versions 3 and 4. The second edition, *Professional Apache Tomcat 5,* focused primarily on Tomcat 5. Since then, Tomcat has released a new edition, and hence the need for this book.

What's Changed Since the Second Edition

Those of you who own a copy of our previous book will no doubt be wondering what's changed in this one, and if it justifies purchasing an updated version.

Well, a lot has changed — and improved! There is a new specification (Servlet 2.5, JavaServer Pages 2.1) and a brand-new Tomcat version (Tomcat 6) implementing it. Tomcat 6 boasts of performance and memory optimizations, faster and more scalable Connectors, and an improved clustering implementation.

Other than updated content, you will find the following in the book:

- Complete and updated coverage for Tomcat 6: This book focuses exclusively on the new Tomcat version.
- *Performance, Performance, Performance*: Tomcat has finally come into its own, and is no longer a developer's stepping stone to a more "industrial strength" server. Its use by a veritable Who's Who of Fortune 500 companies, as well as highly trafficked Web sites, attests to this. The book reflects this status by adding a new chapter on performance tuning as well as coverage of the new, high-performance APR and NIO Connectors.
- A new chapter on logging: Both Tomcat server logs as well as logging from Web applications. The chapter also covers log file management strategies and log analysis.
- An enhanced chapter on managing and monitoring Tomcat using its JMX support.
- A reworked chapter on clustering: Tomcat 6 introduces improvements in its clustering support, including a new clustering configuration.
- A reworked chapter on securing Tomcat installations and Web applications.
- Coverage of the Web server Connectors for Tomcat 6 — `mod_proxy` and `mod_jk`.
- And many other topics!

We value your feedback, and have improved on areas that needed some changes in our second edition. You will find several of our original chapters rewritten based on your suggestions, with better organization and more content.

How to Use This Book

The best way to read a book is from cover to cover. We do recognize, however, that for a technical book of this nature, it is often not possible to do that. This is especially true if a busy administrator wants to refer to this book only for a particular urgent task at hand.

We have written this book to address both needs.

The chapters are structured so that they can be read one after another, with logically flowing content. The chapters are also independent to the degree possible, and include references to other sections in the book when it is necessary to have an understanding of some background material first.

This book is organized as follows:

- **Chapter 1, "Apache Tomcat,"** provides an introduction to the Apache and Tomcat projects, their history, and information about the copyright licenses under which they can be used.
- **Chapter 2, "Web Applications: Servlets, JSPs, and More,"** is a "10,000-foot overview" of Web technologies for administrators unfamiliar with them, including CGI, servlets, JSPs, JSP tag libraries, and MVC (Model-View-Controller) architecture.
- **Chapter 3, "Tomcat Installation,"** details the installation of JVM and Tomcat on Windows and Unix/Linux systems, and offers troubleshooting tips.
- **Chapter 4, "Tomcat Architecture**," provides a conceptual background on components of the Tomcat 6 server architecture, including Connectors, Engines, Realms, Valves, Loggers, Hosts, and Contexts.
- **Chapter 5, "Basic Tomcat Configuration,"** covers the configuration of the Tomcat server components introduced in Chapter 4.
- **Chapter 6, "Advanced Tomcat Features,"** details advanced Tomcat configuration topics, such as access log administration, single sign-on across Web applications, request filtering, the Persistent Session Manager, and JavaMail session setup.
- **Chapter 7, "Web Application Configuration,"** describes the structure of Web applications deployed in Tomcat, and their configurable elements.
- **Chapter 8, "Web Application Administration,"** explains how these Web applications can be packaged, deployed, undeployed, and, in general, managed. There are three ways to do this in Tomcat: via HTTP commands, via a Web-based GUI, and through Ant scripts. This chapter describes all of them.
- **Chapter 9, "Class Loaders,"** introduces Java class loaders and discusses their implications for Tomcat, including (but not limited to) security issues.
- **Chapter 10, "HTTP Connectors,"** describes Tomcat's internal HTTP protocol stack that enables it to work as a Web server. The chapter covers its configuration, as well as security and performance issues.

- **Chapter 11, "Tomcat and Apache HTTP Server,"** covers the use of Apache as a Web server frontend for Tomcat using both Apache's `mod_proxy` as well as the JK Connector. It also describes load-balancing configurations, as well as SSL setup.
- **Chapter 12, "Tomcat and IIS,"** provides detailed coverage of the use of IIS as a Web server frontend for Tomcat.
- **Chapter 13, "JDBC Connectivity,"** discusses JDBC-related issues in Tomcat, such as connection pooling, JNDI emulation, configuring a data source, and alternative JDBC configurations.
- **Chapter 14, "Tomcat Security,"** deals with a wide range of security issues, from securing Tomcat installations to configuring security policies for Web applications that run on it.
- **Chapter 15, "Shared Tomcat Hosting,"** will prove very useful to ISPs and their administrators, as it covers Tomcat installations in virtual hosting situations.
- **Chapter 16, "Monitoring and Managing Tomcat with JMX,"** explores Tomcat's Java Management Extension (JMX) support in detail.
- **Chapter 17, "Clustering,"** covers Tomcat configurations for providing scalability and high availability to Web applications. This is a "must read" chapter for production deployments of Tomcat.
- **Chapter 18, "Embedded Tomcat,"** details the mechanism for embedding Tomcat within custom applications.
- **Chapter 19, "Logging,"** covers logging by the Tomcat server and Web applications, and techniques for log file management and log analysis.
- **Chapter 20, "Performance Testing,"** explains how to develop a performance test plan for Web applications, and how to do performance test using the open-source JMeter framework.
- **Chapter 21, "Performance Tuning,"** suggests where and how to look for the root cause when faced with specific Tomcat performance issues. This chapter also covers performance tuning tips and best practices for Tomcat 6.
- **Appendix A, "Tomcat and IDEs,"** covers the support available for Tomcat in two popular open source IDEs: Eclipse and NetBeans.
- **Appendix B, "Apache Ant,"** provides a tutorial introduction to Ant, as well as solutions for common tasks that system administrators need to do while developing build and deploy scripts. Apache Ant is used extensively in the book, both as a build/install tool, as well as a scripting engine. Ant is the standard tool used by administrators to automate repetitive tasks for Java-based Web development.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.

Tips, hints, tricks, and cautions regarding the current discussion are offset and placed in italics like this.

As for styles in the text:

- New and defined terms are highlighted in *italics* when first introduced.
- Keyboard strokes appear as follows: Ctrl+A.
- Filenames, URLs, directories, utilities, parameters, and other code-related terms within the text are presented as follows: `persistence.properties`.
- Code is presented in two different ways:

```
In code examples, we highlight new and important code with a gray background.
The gray highlighting is not used for code that's less important in the given
context or for code that has been shown before.
```

Downloads for the Book

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at `wrox.com/WileyCDA/WroxTitle/productCd-0471753612.html`. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-471-75361-2.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at `wrox.com` to see the code available for this book and all other Wrox books.

Errata

We made every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, such as a spelling mistake or a faulty piece of code, we would be very grateful for your feedback. By sending us errata, you may save other readers hours of frustration, and you will be helping to provide even higher quality information.

To find the errata page for this book, go to `wrox.com` and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page, you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list, including links to each book's errata, is also available at `wrox.com/misc-pages/booklist.shtml`.

If you don't spot the error you found on the Book Errata page, go to `wrox.com/contact/techsupport.shtml` and complete the form that is provided to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in a subsequent edition of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at `http://p2p.wrox.com`. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature if you wish to be sent e-mail about topics of particular interest to you when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At the P2P Web site, you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `http://p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail message with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages that other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

Caveat

Finally, a caveat: Tomcat, like all active open-source projects, is a constantly evolving piece of software. This is usually good, because it keeps the software abreast of new technologies and improves existing ones. However, this can make the content in any related book outdated over time. This is especially true of new features that have been added in Tomcat 6. While we have made every effort possible to ensure that the book remains current, we would like to point you to the following additional resources:

- **Book Errata:** Any changes in the book caused by new (or modified) Tomcat features will be posted in the book errata section of the Wrox Web site (`wrox.com`) under the Book List link.
- **Wrox P2P forum (`http://p2p.wrox.com`):** The place where you can consult with the Wrox user community.
- **Tomcat User's mailing list:** Mailing list for Tomcat users. This is where questions relating to Tomcat's usage and configuration should be posted. The archives for the list are at `http://mail-archives.apache.org/mod_mbox/tomcat-user/` and `http://marc.theaimsgroup.com/?l=tomcat-user`, and directions for joining the list are at `http://tomcat.apache.org/lists.html`.

- **Tomcat Developer's mailing list:** Mailing list for developers of the Tomcat Servlet container. This is the place to track new developments in Tomcat. *Do not post user questions on this list; use the Tomcat User's mailing list instead.* The archives for the list are at `http://mail-archives.apache.org/mod_mbox/tomcat-dev/` and `http://marc.theaimsgroup.com/?l=tomcat-dev`, and directions for joining the list are at `http://tomcat.apache.org/lists.html`.
- Yet another place to monitor Tomcat developments is the IRC channel `http://tomcat.apache.org/irc.html`.
- **The Apache bug database:** Apache uses a Bugzilla-based system to track bugs (`http://issues.apache.org/bugzilla/`). This is where (using the Query Existing Bug Reports option in Bugzilla) you can verify whether the issue you are facing is configuration-related or a known Tomcat bug.

Professional
Apache Tomcat 6

1

Apache Tomcat

If you've written any Java servlets or JavaServer Pages (JSPs), chances are good that you've downloaded Tomcat. That is because Tomcat is a free, feature-complete *Servlet container* that developers of servlets and JSPs can use to run their code. Tomcat is used in Sun's reference implementation of the Servlet Container, which means that Tomcat's first goal is to be 100 percent compliant with the versions of the Servlet and JSP API specifications that it supports.

However, Tomcat is more than just a test server. Many corporations are using Tomcat in production environments because it has proven to be quite stable. These corporations range from Fortune 500 companies such as WalMart and General Motors to ISPs hosting multiple small-business Web sites. Tomcat is used in the real world to run everything from online photo albums (Webshots) to high performance financial Web applications (ETrade).

A list of Tomcat-powered Web sites is at `http://wiki.apache.org/tomcat/PoweredBy`.

Despite Tomcat's popularity, it suffers from a common shortcoming among open source projects: lack of complete documentation. Some documentation is distributed with Tomcat (mirrored at `http://tomcat.apache.org`), and there's an open source effort to write a Tomcat book (`http://tomcatbook.sourceforge.net/`). Even with these resources, however, there is a great need for additional material.

This book has been created not just to fill in some of the documentation holes, but to use the combined experience of the authors to help Java developers and system administrators make the most of the Tomcat product. Whether you're trying to learn enough to just get started developing Web applications or want to understand the more arcane aspects of Tomcat configuration, you should find what you're looking for within these pages.

The first two chapters are designed to provide newcomers with some basic background information that is prerequisite learning for subsequent chapters. If you're a system administrator with no previous Java experience, we advise you to read these first two chapters, and likewise if you're a Java developer who is new to Tomcat. If you're well informed about Tomcat and Java, you'll

probably want to jump straight ahead to Chapter 3, although skimming this chapter and its successor is likely to add to your present understanding.

The following topics are discussed in this chapter:

- The origins of the Tomcat server
- The terms of Tomcat's license and how it compares to other open source licenses
- How Tomcat fits into the Java "big picture"
- An overview of integrating Tomcat with Apache and other Web servers

Humble Beginnings: The Apache Project

One of the earliest Web servers was developed by Rob McCool at the National Center for Supercomputer Applications (NCSA), University of Illinois, Urbana-Champaign. This Web server was referred to colloquially as the NCSA project, or NCSA for short. By 1995, the NCSA server was quite popular, but its future was uncertain because the primary developer, McCool, had left NCSA the previous year. A group of developers got together and compiled all the NCSA bug fixes and enhancements they had found, and patched them into the NCSA code base. The developers released this new version in April 1995, and called it Apache, which was somewhat of an acronym for "A PAtCHy Web Server."

Apache was readily accepted by the developer community from its earliest days, and less than a year after its release, it unseated NCSA to become the most used Web server in the world (measured by the total number of servers running Apache), a distinction that it has held ever since (according to Apache's Web site). Incidentally, during the same period that Apache's use was spreading, NCSA's popularity was plummeting, and by 1999, NCSA was officially discontinued by its maintainers.

> *For more information on the history of Apache and its developers, see* `http://httpd.apache.org/ABOUT_APACHE.html`.

Today, the Apache Web server is available on just about any major operating system (in addition to the source code download, Apache binaries are available for over a dozen operating systems). Apache can be found running on some of the largest server farms in the world, as well as on some of the smallest devices (including several hand-held devices). In UNIX data centers, Apache is as ubiquitous as air conditioning and UPS systems.

While Apache was originally a somewhat mangy collection of miscellaneous patches, today's versions are rock-solid production quality servers. The only real competitor to Apache in terms of market share and feature set is Microsoft's Internet Information Server (IIS), which is bundled free with certain versions of the Windows operating system. As of this writing, Apache's market share is estimated at around 60 percent, with IIS at 30 percent (statistics courtesy of `http://news.netcraft.com/archives/web_server_survey.html`).

It is also worth noting that Apache has a reputation for being much more secure than Microsoft IIS. When new vulnerabilities are discovered in either server, the Apache developers fix Apache far faster than Microsoft fixes IIS.

The Apache Software Foundation

In 1999, the same folks who wrote the Apache server formed the Apache Software Foundation (ASF). The ASF is a nonprofit organization that was created to facilitate the development of open source software projects. Tomcat is developed under the auspices of the ASF. According to their Web site, the ASF accomplishes this goal by doing the following:

- Providing a *foundation* for open, collaborative software development projects by supplying hardware, communication, and business infrastructure
- Creating an independent legal entity to which companies and individuals can *donate resources* and be assured that those resources will be used for the public benefit
- Providing a means for individual volunteers to be sheltered from *legal suits* directed at ASF projects
- Protecting the Apache *brand* (as applied to its software products) from being abused by other organizations

In practice, the ASF does indeed sponsor a great many open source projects. While the best-known of these projects is likely the aforementioned Apache Web server, the ASF hosts many other well-respected and widely used projects, including such respected industry standards as the following:

- **Xerces:** A Java/C++ XML parser with JAXP bindings
- **Ant:** A Java-based build system (and much more)
- **Axis:** A Java-based Web services implementation

The number of ASF-sponsored projects is growing fast. Visit `www.apache.org` to see the latest list.

Tomcat

The Tomcat project has its origins in the earliest days of Java's servlet technology. *Servlets* are a certain type of Java application that plug into special Web servers, called *Servlet containers* (originally called Servlet engines). Sun created the first Servlet container, called the Java Web Server, which demonstrated the technology but wasn't terribly robust. Meanwhile, the ASF folks created the JServ product, which was a Servlet engine that integrated with the Apache Web server.

In 1999, Sun donated its Servlet container code to the ASF, and the two projects were merged to create the Tomcat server. Today, Tomcat is used by Sun in its *reference implementation* (RI), which means that Tomcat's first priority is to be fully compliant with the Servlet and *JavaServer Pages* (JSP) specifications published by Sun. This is discussed in more detail in Chapter 2.

The first version of Tomcat was the 3.*x* series, and it implemented the Servlet 2.2 and JSP 1.1 specifications. The Tomcat 3.*x* series was descended from the original code that Sun provided to the ASF in 1999.

In 2001, Tomcat 4.0 (code-named Catalina) was released. Catalina was a complete redesign of the Tomcat architecture, and built on a new code base. The Tomcat 4.*x* series was used in the RI of the Servlet 2.3 and JSP 1.2 specifications.

The latest version of Tomcat, Tomcat 6, implements the Servlet 2.5 and JSP 2.1 specifications. In addition, it boasts of an improved clustering implementation over the previous iteration (Tomcat 5.5).

> **Tomcat used to be a subproject under the Apache Jakarta project. The Jakarta project is an umbrella under which the ASF sponsors the development of many Java sub-projects, such as JMeter, Log4j, and Struts. However, Tomcat has now been promoted to a top-level project.**

Distributing Tomcat: The Apache License

Tomcat is open source software, and, as such, is free and freely distributable. However, if you have much experience in dealing with open source software, you're probably aware that the terms of distribution can vary from project to project.

Most open source software is released with an accompanying license that states what may and may not be done to the software. At least 40 different open source licenses are in use, each of which has slightly different terms.

Providing a primer on all of the various open source licenses is beyond the scope of this chapter, but the license governing Tomcat is discussed here and compared with a few of the more popular open source licenses.

Tomcat is distributed under the Apache License, which is listed at `apache.org/licenses`. The key points of this license state the following:

- The Apache License must be included with any redistribution of Tomcat's source code or binaries.
- Any documentation included with redistribution must give a nod to the ASF.
- Products derived from the Tomcat source code can't use the terms "Tomcat," "The Jakarta Project," "Apache," or "Apache Software Foundation" to endorse or promote their software without prior written permission from the ASF.
- Tomcat has no warranty of any kind.

However, through omission, the license contains the following additional implicit permissions:

- Tomcat can be used by any entity (commercial or noncommercial) for free without limitation.
- Those that make modifications to Tomcat and distribute their modified version do not have to include the source code of their modifications.
- Those who make modifications to Tomcat do not have to donate their modifications to the ASF.

Thus, you're free to deploy Tomcat in your company in any way you see fit. It can be your production Web server or your test Servlet container used by your developers. You can also redistribute Tomcat with any commercial application that you may be selling, provided that you include the license and give credit to the ASF. You can even use the Tomcat source code as the foundation for your own commercial product.

Comparison with Other Licenses

Among the previously mentioned and rather large group of other open source licenses, two licenses are particularly popular at the present time: the GNU General Public License (GPL) and the GNU Lesser General Public License (LGPL). Let's take a look at how each of these licenses compares to the Apache License.

GPL

The GNU Project created and actively evangelizes the GPL. The GNU Project is somewhat similar to the ASF, with the exception that the GNU Project would like all of the non-free (that is, closed source or proprietary) software in the world to become free. The ASF has no such (stated) desire and simply wants to provide free software.

Free software can mean one of two entirely different things: software that doesn't cost anything and software that can be freely copied, distributed, and modified by anyone (thus, the source code is included or is easily accessible). Such software can be distributed either free or for a fee. A simpler way to explain the difference between these two types of free is to compare "free," as in "free beer," and "free," as in "free speech." The GNU Project's goal is to create free software of the latter category. All uses of the phrase "free software" in the remainder of this section use this definition.

The differences between the Apache License and the GPL thus mirror the distinct philosophies of the two organizations. Specifically, the GPL has the following key differences from the Apache License:

- No "non-free" software may contain GPL-licensed products or use GPL-licensed source code. If non-free software is found to contain GPL-licensed binaries or code, it must remove such elements or become free software itself.
- All modifications made to GPL-licensed products must be released as free software if the modifications are also publicly released.

These two differences have huge implications for commercial enterprises. If Tomcat were licensed under the GPL, any product that contained Tomcat would also have to be free software.

Furthermore, while the Apache License permits an organization to make modifications to Tomcat and sell it under a different name as a closed source product, the GPL would not allow any such act to occur; the new derived product would also have to be released as free software.

LGPL

The GNU Lesser General Public License (LGPL) is similar to the GPL, with one major difference: Non-free software may contain LGPL-licensed products. The LGPL license is commonly referred to as the "library" GLP because it is intended primarily for software libraries that are themselves free software, but whose authors want them to be available for use by companies who produce non-free software.

If Tomcat were licensed under the LGPL, it could be embedded in non-free software, but Tomcat could not itself be modified and released as a non-free software product.

For more information on the GPL and LGPL licenses, see `www.gnu.org`.

Other Licenses

Understanding and comparing open source licenses can be a rather complex task. The preceding explanations are an attempt to simplify the issues. For more detailed information on these and other licenses, the following two resources can help you:

- The Open Source Initiative (OSI) maintains a database of open source licenses. Visit them at `www.opensource.org`.
- The GNU Project has an extensive comparison of open source licenses with the GPL license. See it at `www.gnu.org/licenses/license-list.html`.

The Big Picture: Java EE

As a Servlet container, Tomcat is a key component of a larger set of standards collectively referred to as the Java Enterprise Edition (*Java EE*) platform. The Java EE standard defines a group of Java-based *APIs* that are suited to creating Web applications for *enterprises* (that is, large companies). To be sure, companies of any size can take advantage of Java EE, but many Java EE technologies are especially designed to solve the problems associated with the creation of large software systems.

Java EE is built on the Java Standard Edition (*Java SE*), which includes the Java binaries (such as the JVM and bytecode compiler), as well as the core Java code libraries. Java EE depends on Java SE to function. Both the Java SE and Java EE can be obtained from `http://java.sun.com`. Both Java SE and Java EE are referred to as *platforms*, because they provide core functionality that acts as a sort of platform or foundation upon which applications can be built.

Since the middle of 2005, Sun has been re-branding some of the Java platform names. Java Enterprise Edition, previously called J2EE, is now called Java EE. Java Standard Edition, previously called J2SE, is now Java SE. Similarly, the mobile edition (previously J2ME) has been renamed to Java ME.

Java APIs

As mentioned, Java EE is a standardized collection of Java APIs. The term *API* (or *application programming interface*) is used by software developers in general to describe services made available to applications by an underlying service provider (such as an operating system). In the Java world, this term is used to describe many of the services that the Java Virtual Machine (JVM) and its code libraries make available to Java programs.

An important characteristic of APIs is that they are separated from the services that provide them. In other words, an API is a kind of technical contract defining the functionality that two parties must provide: a service provider (often called an *implementation*) and an application. If both parties adhere to the contract, an API is *pluggable* (that is, a new service provider can be plugged into the relationship). Of course, if a service provider fails to conform to the contract, the applications that use the API will fail to function properly.

The Java Community Process

APIs in the Java world are created and modified by a standards body known as the Java Community Process (*JCP*). The JCP is composed of hundreds of *Java Specification Requests (JSRs)*. Each JSR is a request to either change an existing aspect of Java (including its APIs) or introduce a new API or feature to Java. New JSRs can be submitted by a *member* of the JCP. Anyone can become a member of the JCP and, notably, individuals may do so at no cost (organizations pay a nominal fee). Once submitted, the JCP *Executive Committee* must approve the JSR. The Executive Committee consists of JCP members who have been elected to three-year terms in an annual election.

When a JSR is approved, the submitter becomes the *Spec Lead*. The Spec Lead forms an *Expert Group* composed of JCP members who assist the Spec Lead in creating a specification detailing the change or addition to the Java language. The Expert Group shepherds the specification along through various review processes (to other JCP members and to the public) until, finally, the JSR is judged completed and is approved by the Executive Committee. If a JSR results in an API, the Expert Group must also provide a reference implementation of the API (discussed earlier in this chapter in the context of Tomcat) and a *technology compatibility kit* (TCK) that other implementers can use to verify compatibility with the API.

Thus, via the JCP, any Java developer can influence the Java platforms, by submitting a JSR, becoming a member of an existing JSR's Expert Group, or by simply giving feedback to JSR Expert Groups. While not the first attempt to create a technology standards body, the JCP is probably the world's best combination of accessibility and influence. As a contrast, the influential World Wide Web Consortium (W3C) standards body charges almost $6,000 for individuals to join. Visit the JCP at `www.jcp.org`.

The Java EE APIs

As mentioned, the Java EE 5 platform consists of many individual APIs. The Servlet and JSP APIs are two of these. The following table describes some of the other Java EE APIs, and a complete list can be found at `http://java.sun.com/javaee/technologies/`.

Java EE API	Description
Enterprise JavaBeans (EJB)	Provides a mechanism that is intended to make it easy for Java developers to use advanced features in their components, such as remote method invocation (RMI), object/relational mapping (that is, saving Java objects to a relational database), distributed transactions across multiple data sources, statefulness, and so on.
Java Message Service (JMS)	Provides high-performance asynchronous messaging. Among other things, it enables Java EE applications to communicate with non-Java systems on top of various transports.
Web service APIs	A set of APIs for Web services and XML processing. These include JAX-WS, JAX-RPC, JAXB, SAAJ, and StAX.
Java Management Extensions (JMX)	Standardizes a mechanism for interactively monitoring and managing applications at runtime.

Table continued on following page

Java EE API	Description
Java Transaction API (JTA)	JTA enables applications to gracefully handle failures in one or more of their components by establishing transactions. During a transaction, multiple events can occur, and if any one of them fails, the state of the application can be rolled back to the way it was before the transaction began. JTA provides the functionality of database-transactions technology across an entire distributed application.
JavaMail	Provides the capability to send and receive e-mail via the industry-standard POP/SMTP/IMAP protocols.

In addition to the Java EE–specific APIs, Java EE applications also rely heavily on Java SE APIs. In fact, over the years, several of the Java EE APIs have been migrated to the Java SE platform. Two such APIs are the Java Naming and Directory Interface (JNDI), used for interfacing with LDAP-compliant directories (and much more), and the Java API for XML Processing (JAXP), which is used for parsing and transforming XML (using XSLT). The vast collection of Java EE and Java SE APIs form a platform for enterprise software development unparalleled in the industry.

Java EE Application Servers

As mentioned, an API simply defines services that a service provider (i.e., the implementation) makes available to applications. Thus, an API without an implementation is useless. While the JCP does provide RIs of all the APIs, using them piecemeal is not the most efficient way to build applications. Enter the *Java EE application server*.

Various third parties provide commercial-grade implementations of the Java EE APIs. These implementations are typically packaged as a Java EE application server. Whereas Tomcat provides an implementation of the Servlet and JSP APIs (and is thus called a Servlet container), application servers provide a superset of Tomcat's functionality: the Servlet and JSP APIs plus all the other Java EE APIs, and some Java SE APIs (such as JNDI).

Dozens of vendors have created *Java EE–compatible* application servers. Being called "Java EE–compliant" means that a vendor of an application server has paid Sun a considerable sum, and has passed various compatibility tests. Such vendors are said to be *Java EE licensees*.

The two most widely used commercial Java EE application servers are Websphere from IBM and Weblogic from BEA. Other than these, there are a number of open source implementations too, such as the following:

- JBoss (`www.jboss.org`)
- JOnAS (`jonas.objectweb.org`)
- Geronimo (`geronimo.apache.org`)
- Glassfish (`glassfish.dev.java.net`)

"Agree on Standards, Compete on Implementation"

Developers who use the Java EE APIs can use a Java EE–compatible application server from any vendor, and it is guaranteed to work with their applications. This flexibility is intended to help customers avoid

vendor lock-in problems, enabling users to enjoy the benefits of a competitive marketplace. The Java slogan along these lines is "Agree on standards, compete on implementation," meaning that the vendors all cooperate in establishing universal Java EE standards (through participation in the JCP) and then work hard to create the best application server implementation of those standards.

That's the theory, at least. In reality, this happy vision of vendor neutrality and open standards is slightly marred by at least two factors. First, each application server is likely to have its own eccentricities and bugs. This leads to a popular variation on the famous "Write Once, Run Anywhere" Java slogan: "Write Once, Test Everywhere." Second, vendors are rarely altruistic. Each application server typically includes a series of powerful features that are outside the scope of the Java EE APIs. Once developers take advantage of these features, their application is no longer portable, resulting in vendor lock-in. Developers must, therefore, be vigilant to maintain their application's portability, if such a capability is desirable.

Tomcat and Application Servers

Up to this point, Tomcat has been referred to as an implementation of the Servlet/JSP APIs (i.e., a Servlet container). However, Tomcat is more than this. It also provides an implementation of the JNDI and JMX APIs. However, Tomcat is not a complete Java EE application server; it doesn't provide support for even a majority of the Java EE APIs.

Interestingly, many application servers actually use Tomcat as their implementation of the Servlet and JSP APIs. Because Tomcat permits developers to embed Tomcat in their applications with only a one-line acknowledgment, many commercial application servers quietly rely on Tomcat without emphasizing that fact. The JBoss and JOnAS application servers mentioned previously make explicit use of Tomcat.

Developers seeking to create Java Web applications that utilize the Servlet, JSP, JNDI, and JMX APIs will find Tomcat an excellent solution. However, those seeking support for additional APIs will probably be better served to either find an application server, or use Tomcat in addition to an application server. A third option is to find an implementation of the individual Java EE APIs required and use them in conjunction with Tomcat. This piecemeal approach is perfectly valid, although integration problems may manifest themselves.

Do you always need a full-fledged Java EE application server to develop enterprise applications? The short answer is "It depends on your requirements." An increasing number of Web sites eschew the traditional Java EE technologies — especially EJB — and develop fairly complex applications with "lightweight" and often open source components. These typically use an application framework such as Struts or Spring, or an object-relational mapping framework such as Hibernate — all running in a state-of-the-art Servlet container, i.e., Tomcat!

Tomcat and Web Servers

Tomcat's purpose is to provide standards-compliant support for Servlets and JSPs. The purpose of Servlets and JSPs is to generate Web content such as HTML files or GIF files on demand, using changing data. Web content that is generated on demand is said to be *dynamic*. Conversely, Web content that never changes and is served up as is, is called *static*. Web applications commonly include a great deal of static content, such as images or Cascading Style Sheets (CSS).

While Tomcat is capable of serving dynamic and static content, many production deployments use a native Web server, such as Apache HTTP Server or IIS, to handle the static content. There are many reasons for choosing to do this, some of which relate to performance and others relate to support of legacy code. Chapters 11 and 12 address these issues in greater detail.

Recognizing that Tomcat could enjoy a synergistic relationship with conventional Web servers, the earliest versions of Tomcat included a "Connector" that enabled a Tomcat and Apache Web server to work together. In such a relationship, Apache receives all of the HTTP requests made to the Web application. Apache then recognizes which requests are intended for Servlets/JSPs, and passes these requests to Tomcat. Tomcat fulfills the request and passes the response back to Apache, which then returns the response to the requestor.

The Apache Connector was initially crucial to the Tomcat 3.*x* series, because Tomcat's support for both static content and its implementation of the HTTP protocol were somewhat limited.

Starting with the 4.*x* series, Tomcat featured a much more complete implementation of HTTP and better support for serving up static content, and should by itself be sufficient for most deployments.

If you're not using either Apache or IIS or any other Web server officially supported by Tomcat, then don't give up hope entirely. It is still very possible to integrate Tomcat with other Web servers, even one that resides on the same machine. If you wish to run them on the same machine for instance, all you have to do is to set up Tomcat and the Web server to run on different network ports. You then can then design your Web application to request its static resources from the Web server, and have Tomcat handle the requests for dynamic content.

> **In many situations it might be simpler to just use Tomcat's own Web server implementation. Tomcat has an "HTTP Connector" — i.e., a component that implements an HTTP server. More on this, including when it makes sense to use this, and when a native Web server is a better choice, is explained in Chapter 10.**

Summary

To conclude this chapter overview of Tomcat, let's review some of the key points we discussed:

- The Apache Software Foundation (ASF) is a nonprofit organization created to provide the world with quality open source software.
- The ASF maintains an extensive collection of open source projects. Many of the ASF's Java projects are collected under the umbrella of a parent project called Jakarta.
- Tomcat started as a subproject of the Jakarta project, but now is independent of it.
- Tomcat can be freely used in any organization. It can be freely redistributed in any commercial project so long as its license is also included with the redistribution and proper recognition is given.

- Java EE is a series of Java APIs designed to facilitate the creation of complex enterprise applications. Java EE–compatible application servers provide implementations of the Java EE APIs.
- Tomcat is a Java EE–compliant Servlet container and is the official reference implementation for the Java Servlet and JavaServer Pages APIs. Tomcat also includes implementations of the JNDI and JMX APIs, but not the rest of the Java EE APIs, and is not, thus, a complete Java EE application server.
- While Tomcat can function as a Web server, it can also be integrated with other Web servers.
- Tomcat has special support for integrating with the Apache, IIS, and Netscape Enterprise Server (NES) servers, among others.

This chapter has provided a basic introduction to Tomcat. Chapter 2 describes what Tomcat-served Web applications look like and what files they comprise. It also provides a quick background to Web applications, which should be useful for administrators who do not have a background in Java technologies.

2

Web Applications: Servlets, JSPs, and More

Tomcat administrators do not need to be Java or Web developers; however, an understanding of the technologies involved is useful. Toward that objective, this chapter provides an introduction to the following technologies for building dynamic Web sites:

- CGI scripts
- Server-based Java technologies: servlets, JSPs, and tag libraries
- MVC architecture, and implementations (Struts, for example)
- Web applications built using the preceding technologies

A Brief History of Web Applications

Initial Web sites were static — they merely served up HTML pages. Also, the protocol for serving up Web pages, *Hypertext Transfer Protocol* (*HTTP*), was a simple, *stateless* protocol.

This state of affairs didn't last long. Soon, there was a new need for showing information that changed with time. Also, people wanted to do more complex things with Web sites, for example keep track of what the user did the last time, so as to enable more complex commerce related interactions, such as putting items in a shopping cart. The first required a mechanism to serve up dynamic content, and the second required a way to maintain state over a stateless protocol such as HTTP.

CGI Scripts: The First Mechanism for Dynamic Content

The first mechanism for serving up dynamic content to users was the *Common Gateway Interface* (*CGI*). Executable applications (usually, but not necessarily, written in PERL or C) were provided with an interface that enabled clients to access them in a standard way over HTTP.

More details on CGI can be found at the W3C (World Wide Web Consortium) Web page at `www.w3.org/CGI/`.

A URL for a CGI program looks something like this fictitious URL:

```
http://www.myserver.com/cgi-bin/MyExecutable?name1=value1&name2=value2
```

The first part of the URL is the *protocol name* (in this case HTTP), followed by the name of the server. Everything after this and before the question mark is the *context path*.

The `/cgi-bin/` part of the URL alerts the server that it should execute the CGI program specified in the next part of the URL, in this case `MyExecutable`. The section after the question mark is known as the *query string*, and it enables the client to send information to the CGI program. In this way, the program can run with client-specific information affecting the results.

CGI suffered from several drawbacks:

- Each incoming CGI request required starting an operating system process.
- This process would then load and run a (CGI) program.
- Tedious and repetitive coding was needed to handle the network protocol and request decoding.

The first two operations in the previous list can use a large number of CPU cycles and a lot of memory. Because both operations must be performed for each request, a server machine could get overloaded if too many requests arrive in a short period of time. Also, because each CGI program is independent of the other, and often in incompatible programming languages, it is not possible to share code used in networking and request decoding.

Note that CGI describes only the contract between the Web server and the program. No services are provided to help implement user-centric systems. Thus it is hard to do things such as maintain the identity of the client across requests, restrict access to the application to authorized users, and store runtime information in the application. For this reason, CGI scripts are not used by most modern Web sites. You might still come across some being used for simple tasks such as processing forms.

Hence, it became necessary to provide a framework for building Web applications. This framework would provide the services mentioned previously, in addition to solving the problem of poor performance and scalability.

Server Side Java: Servlets

Servlets are portions of logic written in Java that have a defined form and are invoked to dynamically generate content or perform some action.

Servlets overcome the problems mentioned earlier with CGI programs, such as the following:

- The overhead of starting an operating system process for each request is eliminated. A Java virtual machine (JVM) is kept running, and all requests are handled by that JVM.
- Java classes are loaded by the JVM to process incoming requests; if more than one request requires the same processing, the already loaded class can be used to handle it. This eliminates the class loading overhead for all but the first request.

- The issue of managing state over a stateless protocol such as HTTP is solved, as you will see later in the chapter.
- Code that handles the networking protocol and decodes incoming requests can be shared by all the request-processing Java classes.

The Servlet Interface

Servlets have a defined form: This means that all servlets implement an interface called `Servlet`, which defines a standard lifecycle, i.e., a list of methods that are called in a predictable way. Initialization is facilitated through a method called `init()`. Any resources needed by the servlet, along with any initialization that the servlet must do before it can service client requests, are implemented by this method, which is called just once for each instance of the servlet.

Each servlet may handle many requests from many clients. The `Servlet` interface defines a method called `service()` that is called for each client request. This method controls the computation and generation of the response that is returned to the client. When a request has been serviced and the response returned to the client, the servlet waits for the next request. The `service()` method checks which type of HTTP request was made (whether `GET` or `POST`, and so on), and forwards the request to methods defined for handling these requests.

Finally, a method called `destroy()` is called once before the `Servlet` class is disposed of (see Figure 2-1). This method can be used to free any resources acquired in the `init()` method.

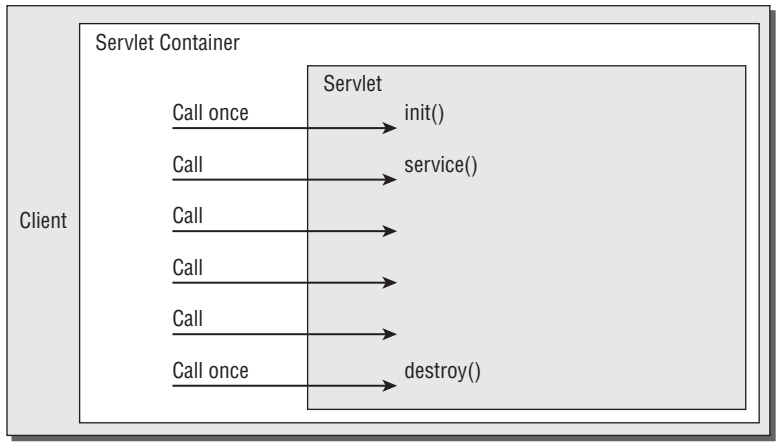

Figure 2-1: Servlet methods

Servlet Containers

Many vendors develop execution environments for servlets known as *Servlet containers*. However, they must ensure that they follow the contract defined for servlets in the Servlet specifications. Therefore, a servlet written according to these specifications should run without modification in any compliant Servlet container.

Depending on the Web server and Servlet container used, very different configurations are possible. Some popular configurations include:

- The same JVM runs the Web server as well as the servlets; in this case the Web server is coded in Java. This was the case for the very first Servlet container implemented by Sun, unimaginatively named Java Web Server. This is often called the *standalone configuration*. Figure 2-2a illustrates the standalone configuration.
- The Web server is not written in the Java programming language, but it starts a Java VM within the same operating system process; in this case, the information is passed directly from the Web server into the Java VM hosting the servlet container and some versions of both the Apache and Microsoft IIS Web servers can work in this way. This is often called the *in-process configuration*. Figure 2-2b illustrates the in-process configuration.
- The Web server is not written in the Java programming language and runs in a separate operating system process from the servlet container; in this case, the Web server passes the request to the servlet container using either a local network or operating system–specific inter-process communications mechanism, and this is the typical configuration for the Apache or Microsoft IIS Web server. This is often called the *independent configuration* or *networked configuration*. Figure 2-2c illustrates this configuration.

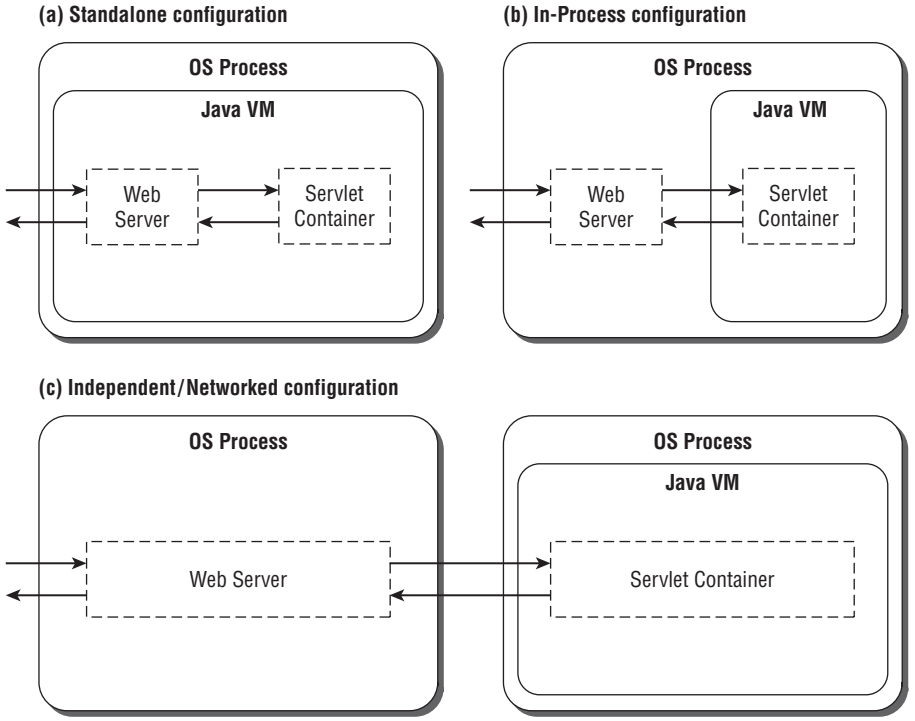

Figure 2-2: Web server and Servlet container configurations

Furthermore, containers provide services in addition to life-cycle management. These include making initialization parameters available, enabling database connections, and enabling the servlet to find and execute other resources in the application. Containers can also maintain a session for the servlet. As

mentioned earlier in the chapter, HTTP is stateless by design. Once the response is returned to the client, there is nothing in HTTP that enables the server to recognize the client when it makes another request.

To solve this problem, the container maintains the client's identity through temporary *cookies* that store a special token referencing the user. This token is known as the *user's session*. By doing this, the container can identify a client to the servlet across multiple requests. This enables more complex interactions with the client.

If cookies are unavailable, the container can also rewrite links in the HTML that is returned to the client, which offers an alternative way to maintain session information. This mechanism is called *URL rewriting*. This means that instead of the application setting cookies in the client browser (and then failing if cookies are disabled) the container automatically determines whether cookies are enabled. If so, it uses them, or alternatively uses URL rewriting to maintain the session. The application developer can then create objects that are stored in the user's session and which are available to other servlets in subsequent client requests.

Security (that is, authentication and authorization) is provided in servlet containers through a *declarative security framework*. This means that restricted resources and authorized users are not hard-coded into the application. Instead, a configuration document specifies the types of users to whom the resources are available. Thus, security policies can be changed easily according to requirements. This is covered extensively in Chapter 14.

Tomcat is one such Servlet container. It provides an execution environment for servlets, provides them with access to system resources (such as the file system), and maintains the client's identity. As mentioned in Chapter 1, it is also used in the reference implementation of the Servlet specifications.

Although the Servlet specifications allow for other transports besides HTTP, in practice, servlets are almost exclusively used to provide application functionality across the Internet, servicing HTTP requests. Like CGI, the Servlet specifications were designed to provide a standard way of extending Web servers beyond static content and creating Web-enabled applications. Unlike CGI, the Servlet specifications are confined to the Java language, although this carries with it the benefits of platform-independence.

Like the Java language, the Servlet specifications were created with the purpose of enabling third parties to offer containers that compete on price, performance, and ease of use. In principle, because these containers are standard, customers of these third parties are free to choose among them and can enjoy a relatively painless migration.

In practice, however, the vendors of Servlet containers also compete with services that exceed the specifications. In addition, there are several areas in which the exact way to implement the specifications is open to interpretation. One example of this is the way in which class loaders (responsible for making classes available within the container so that they can be used by the application) work within the container. Tomcat's class loaders are covered in Chapter 9.

However, migration is usually more a container configuration issue than a matter of reprogramming and recompiling the application. This assumes, however, that the programmers were not tempted into using nonstandard services of the Servlet container, and programmed the application with cross-container compatibility in mind.

> *Tomcat, as a result of its reference implementation status, does not provide extra-specification features that create application dependencies on it.*

Accessing Servlets

If you consider servlets as program resources, how are these resources accessed? Well, like CGI, the server maps URLs to programmatic resources.

The recommended way to access servlets is through *logical mapping*, which maps URL "context paths" to servlets. This is often more obvious in its intention than the straight servlet name because it is possible to add information into the path that provides users with a clue as to the intention of the servlet's action. For example, a servlet that loads all available documents to enable administrative procedures may be called `AdminLoaderServlet`, and may be mapped to a context path such as the following:

```
/admin/LoadDocumentsForAdministration
```

thus giving the user a better idea of what is occurring at this point in the application.

The Servlet container intercepts all requests and looks for patterns in the URL that correspond to a specified servlet, invoking the servlet that matches that pattern. For example, all URLs that end with the `.db` extension may be mapped to `com.wrox.db.`*`ServletName`*.

Another possibility is matching a character sequence to a servlet. For example, a system could match all requests that include the character sequence `upload` to an upload manager servlet that manages the uploading process. Thus, in principle, all of the following URLs would invoke this servlet:

```
http://localhost:8080/upload?file=Hello&locationResolver=World
http://localhost:8080/admin/uploadUserDocument/Hello/World/auth
http://localhost:8080/core/Hello.World.upload
```

In older versions of Tomcat, a servlet could also be accessed by its mapped "servlet name," such as `www.server.com/servlet/`*`ServletName`*, or its fully qualified name, such as `www.server.com/servlet/com.wrox.db.`*`ServletName`*.

This approach has been disabled by default in Tomcat 6, although you can still enable it by uncommenting the following lines from `<TOMCAT_HOME>/conf/web.xml`:

```
    <!-- The mapping for the invoker servlet -->
<!--
    <servlet-mapping>
        <servlet-name>invoker</servlet-name>
        <url-pattern>/servlet/*</url-pattern>
    </servlet-mapping>
-->
```

However, this practice is strongly discouraged for security reasons: Web application configuration files may impose security constraints on a servlet, and allowing users to call it directly can be a "back door" into the application.

Drawbacks of Servlets

Although servlets are an improvement over CGI (especially with respect to performance and server load), they too have drawbacks. Their primary use is for processing logic. For presentation of content (i.e., HTML) they are less usable. Hard-coding textual output (including HTML tags) in code makes the

application less maintainable because when text in the HTML must be changed, the servlet must be recompiled. Take a look at an excerpt of servlet code:

```
out.println("<html>");
out.println("  <head>");
out.println("    <title>Hello World example</title>");
out.println("  </head>");
out.println("  <body bgcolor=\"white\">");
out.println("    <h1>Hello World</h1>");
out.println("  </body>");
out.println("</html>");
```

The intended effect of this section of code is to output the following HTML:

```
<html>
  <head>
    <title>Hello World example
  </head>
  <body bgcolor="white">
    <h1>Hello World</h1>
  </body>
</html>
```

This is a rather cumbersome way of doing Web programming.

Second, it requires the HTML designer to understand enough about Java to avoid breaking the servlet. More likely, however, the programmer of the application must take the HTML from the designer and then embed it into the application, which is an error-prone task.

To solve this problem, the *JavaServer Pages (JSP)* technology was created by Sun Microsystems.

JavaServer Pages

The first edition of the JavaServer Pages (JSP) specifications resembled *Active Server Pages (ASP)*, a Microsoft technology. Both have since evolved from those early days so much that the resemblance is now purely superficial. JSP has made a huge leap forward with the introduction of *tag libraries*. These tag libraries are collections of custom tags, and each tag corresponds to a reusable Java module. JSP tags are discussed later in this chapter.

A page coded in JSP could look like the following:

```
<%@ page language="java" %>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <%
    String message = request.getAttribute("message");
    if(message == null || message.equals("")) {
      message = "Hello World";
    }
    %><%=message%>
  </body>
</html>
```

Behind the scenes, the JSP is compiled into a servlet class the first time it is invoked. This servlet is then called for each subsequent request, avoiding the parsing and compiling of the JSP every time a user accesses the site. JSP took off largely as a result of its suitability for creating dynamic visual content at a time when the Internet was growing massively in popularity.

Like servlets, JSPs operate within a container. The JSP container provides the same services as a Servlet container, but requires the additional steps of conversion to servlet code and compilation before the JSP is executed. Tomcat includes both the Servlet container named Catalina that executes servlets and compiled JSPs, and the compiler for JSP files (the Jasper compiler). The combination of a JSP compiler and a Servlet container is known as a Web container (a container capable of hosting Java Web applications).

One practical difference between servlets and JSPs is that servlets are provided in compiled form, whereas JSPs often are not (although pre-compilation is possible). What this means for a system administrator is that servlet files are held in the private resources section of the server, whereas JSP files are mixed in with static HTML pages, images, and other resources in the public section. If good development practices aren't followed, it can often affect the maintainability of a Web site.

Early Web Applications: Model 1 Architecture

Even with the use of servlets and JSPs, a lot of unmaintainable Web applications were built in the early days of Web development. There were two major causes of this: First, the control flow of the Web application (i.e., what content should be shown and in what order) was often coded inside the Web pages themselves. Second, the business logic of the Web site was tightly coupled with the user interface presentation.

These kinds of architectures are today known as *Model 1 architectures*. This architecture is suitable only for small sites with limited functionality, or Web pages with minimal requirements for expansion. It is quite easy to create sites in this way and, therefore, productivity is improved when complexity is low. This model is not recommended for larger sites. The cost of this initial productivity is the time lost in debugging as the complexity and the size of the site increase. Model 1 architecture is illustrated in Figure 2-3.

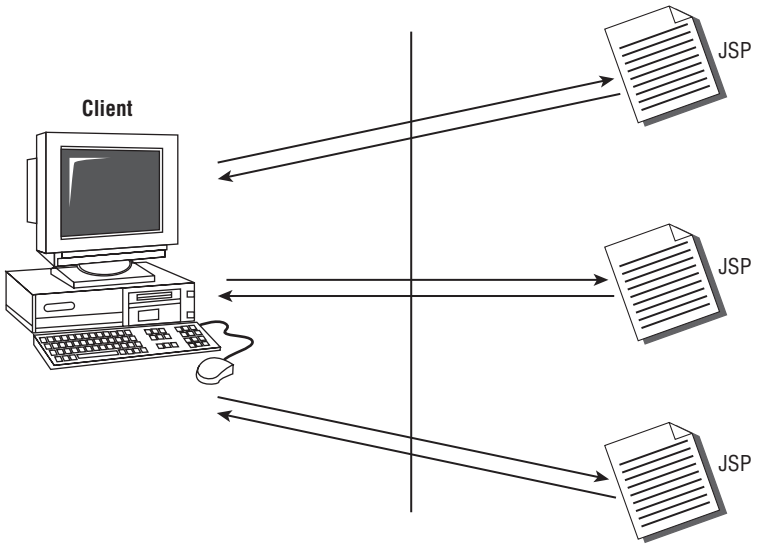

Figure 2-3: Model 1 architecture

As you can see from this diagram, each JSP must know where the user originated from, and where they should be sent next. Thus, in addition to handling the presentation of the Web site — font colors and so on — the JSP also needs to handle the flow of control within the Web application. This approach can soon become unmanageable as the complexity of the Web site increases, and also makes changing the control flow difficult.

The other problem typical to Web sites of this vintage was mixing business logic code with HTML generation, i.e., presentation logic. Mixing code and HTML on the same page means that the designer must be sufficiently proficient with Java code to avoid breaking the functionality of the Web page, as well as be able to work with the logic on the page to produce the desired output. At the same time, the developer must do some of the designer's work of laying out the page when the logic is sufficiently convoluted.

In addition, because pieces of logic may be strewn around the page embedded in sections of HTML, it is by no means straightforward to figure out the intended result without a fair amount of inspection. This can cause significant problems with maintenance, as the code is mixed with markup. Also, the code reusability in such Web sites is very limited, as the same sections of code are often repeated across the site.

The obvious alternative to this is to keep the pages as free from Java as possible, and have the processing logic localized to Java classes.

Modern Web Development: Model 2 Architecture and Web Frameworks

Modern Web development "best practices" include some ground rules for designing Web sites that are easy to maintain and extend:

- Do not embed the logic of handling user requests and the control flow inside the JSP pages themselves. This makes it difficult to maintain the code.
- Do not mix the application logic with the user interface logic (a.k.a. presentation logic).

  The "MVC Architecture" section later in the chapter explains how this is done and mentions some commonly used implementations, such as Apache Struts. This architecture is also sometimes called *Model 2 architecture* to contrast it with Model 1 mentioned earlier.
- Keep Java code (a.k.a. scriptlets) out of JSPs. JSP tag libraries and JSP EL (Expression Language) help developers to do this.
- Another best practice is the use of JSPs as templates. For example, the header that includes the company logo may be in one JSP page, the main menu for the site may be in another, and a current news table may be defined in a third. When the client makes a request, these separate elements of the page are assembled and presented to the user as though they were all created as one, as shown in Figure 2-4. The "Tiles component" of Struts is an example of such a template system. Templating allows for reuse of common portions of Web pages, such as headers, menus, and footers, and allows the designer to effect changes globally by updating a single page.

The technologies that facilitate these modern Web development approaches include JSP tag libraries, JSP Expression Language (EL), and Web frameworks, particularly MVC frameworks.

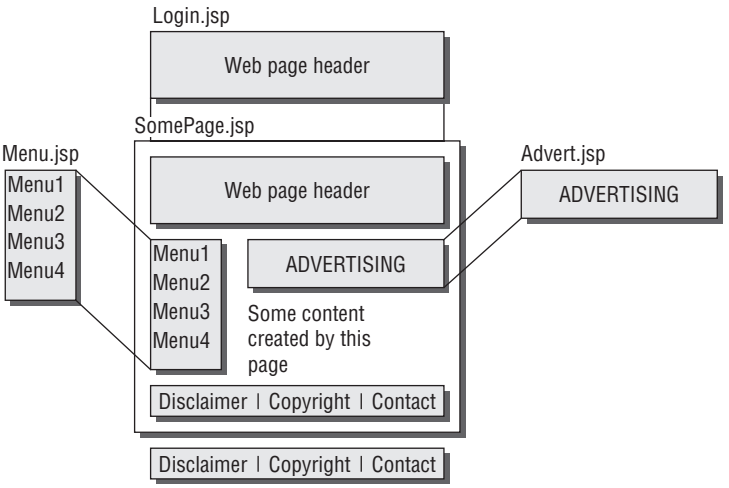

Figure 2-4: Sample structure of a JSP using a template framework

JSP Tag Libraries

JSPs were an improvement over servlets, as far as a mechanism of generating HTML content is concerned. However, JSPs often had a lot of Java code embedded in them. This makes it hard for HTML designers, who often do not have a background as Java developers. Tag libraries help solve this issue to some extent. Note the following JSP, which uses tag libraries:

```
<%@ taglib prefix="app" tagdir="/WEB-INF/tags" %> <html>
  <head>
    <title>Hello World</title>
  </head>
  <body>

    <app:HelloWorld/>

  </body>
</html>
```

Compare this with the previous incarnation of the page:

```
<%@ page language="java" %>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>

    <%
    String message = request.getAttribute("message");
    if(message == null || message.equals("")) {
      message = "Hello World";
    }
    %><%=message%>
  </body>
</html>
```

You can already see that this is an improvement. An HTML-like tag has encapsulated the entire functionality behind our code. In fact, the more complex the application is, the more this replacement of Java code scriptlets with JSP tags improves the readability of the site. The HTML page designer is presented with sections of dynamic content that are far more familiar and, more important, can insert the `HelloWorld` tag without understanding how it works.

Each tag has a corresponding Java class that contains the code that would otherwise appear on the page. In the Web page, the tags look just like HTML, with a start tag followed by an end tag, with optional content:

```
<aTag>Something here</aTag>
```

The tag life cycle includes a method that is called when the start tag is encountered, called `doStartTag()`; a method that is called when the end tag is encountered, called `doEndTag()`; and a method that is called to reset any state (request specific data) in readiness for the next request.

The tag also has power over which parts of the page are parsed by the application. Depending on the behavior of the tag, it can stop the execution of the page, conditionally include its contents, and have its contents evaluated multiple times. You can use this tag as shown here:

```
<app:if cookie="user" value="">
  Please enter your name...
</app:if>
```

The `app:` prefix denotes a group of tags to which this tag belongs. In the preceding example, the contents of the `<app:if>` tag are evaluated if the cookie named `user` has an empty string as its value. In this case, the user is prompted for a name.

Tag libraries thus present an elegant way to write pages that create dynamic content. Another advantage of tag libraries is that they can be reused in many different JSPs.

JSP EL

JSP Expression Language (EL) defines an easy-to-use syntax for accessing Java beans, request/session parameters, and HTTP headers from JSPs. It also has arithmetic, logical, and conditional expressions, thus doing away with the need for any Java scriptlet code inside a JSP.

As an example, the following JSP with Java scriptlets:

```
<b>Your shopping cart has <%=cartBean.getCount()%> items.
```

can now be replaced by the following scriptlet-free JSP code:

```
<b>Your shopping cart has ${cartBean.count} items.
```

MVC Architecture

Earlier in the chapter you saw an example of the Model 1 architecture, in which each JSP must know where the user should be sent next, and from where the user originated. This soon becomes very complicated and hard to maintain for Web sites of sufficient complexity. *Model 2* or *Model View Controller (MVC)* architecture, helps resolve this issue and aids in the separation of application logic and presentation of HTML content. The *Model* is the logic of the site, the rules that determine what is shown, and to whom it is shown. The *View* component of this architecture is naturally the JSPs that display the content that is created. Finally, the *Controller* designates which part of the Model is invoked, and which JSP is used to render the data. Another way to put this is that the Controller defines the structure of the site and the page flow logic. This information is read from a configuration file, and not buried inside Web pages, unlike in Model 1 architectures. Figure 2-5 shows a diagram of the MVC architecture.

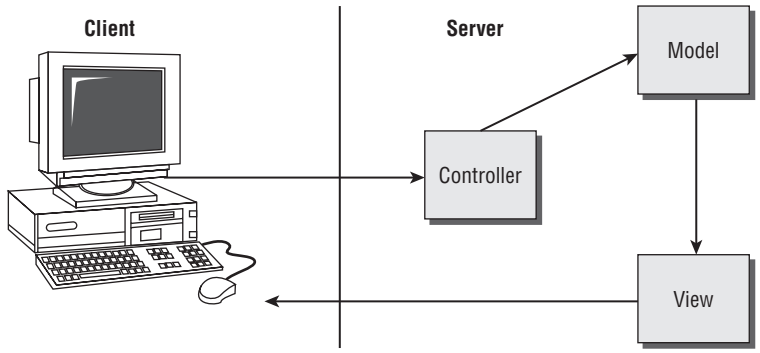

Figure 2-5: MVC architecture

There are two typical types of Model 2 architectures: strict and loose. The strict version designates the role of the Controller to a single servlet, which extracts the information needed to route the query to a piece of logic, executes the logic component, and then forwards the result of the execution to a JSP.

A popularly used MVC implementation is the Struts framework (`http://struts.apache.org/`), which was introduced in Chapter 1. This framework implements a standard servlet for routing execution. Each piece of functionality is implemented as a special type of Struts component known as an *Action*. Each Action defines a single method and can place a variety of objects where the JSP that is invoked can use them to render the page. In this type of architecture, the sequence of execution is often very reliable. Figure 2-6 illustrates this sequence.

Figure 2-6: Sequence of execution in the Struts framework

An expanded example of the MVC strict architecture is shown in Figure 2-7. In this diagram, you can see that the single Controller selects the correct logic to execute, and then forwards the result to the View, which renders the results for the client.

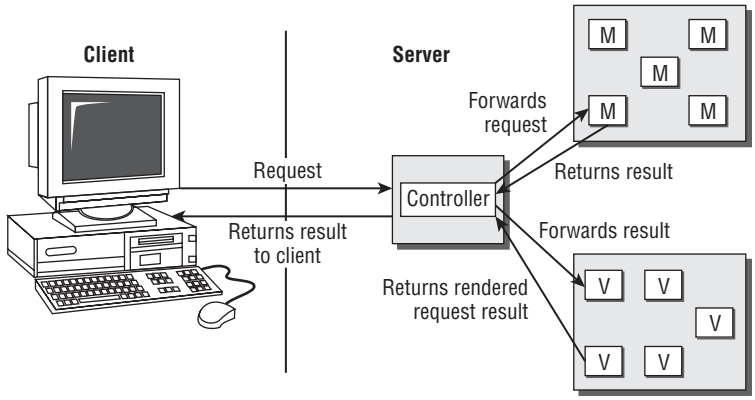

Figure 2-7: Expanded MVC architecture

Small-scale, homegrown sites that are not based on a framework are often a looser version of this architecture. Several servlets each take the role of a Controller and part of the Model. In this version of the Model 2 architecture, the JSPs are still designed so that they contain very little or no logic and the servlets handle all of the functions of the application. This second model is quite popular because it promotes high productivity in the short term, and can be easier to understand than the strict MVC architecture.

In sites that have a pure MVC architecture, the structure of the site is (at least in principle) quite flexible. The site is divided into units of functionality that can be reused in multiple situations, as well as pages that can be reused. For example, a page that displays contact details in a Web site may be used for creating a new contact, updating an old contact, viewing an existing contact, and updating a user's contact details.

A site that allows content management may use the same JSP and servlet code for uploading a variety of documents, each with different needs (such as a report, a tender request, and a procedures manual). As the site expands, these components will need to be integrated with new functionality. The flow control must be configured before the various components will work together correctly.

This does, however, represent a very reusable and updateable site. The site can be reconfigured according to business needs and customer requests with minimal code rewrite.

Using Appropriate Web Technologies

Many other Java technologies are used to build Web applications. A few of these are covered in this book. For instance, JDBC — a Java API used for connecting to database systems — is covered in Chapter 13. If you are interested in more detail about these Java technologies used for building Web applications, you can refer to other books, such as *Beginning JavaServer Pages* (Wrox Press, ISBN 0-7645-7485-X).

Well-designed, modern-day Web applications use all of the server-side Java technologies mentioned, but where they are appropriate:

- Servlets are used to implement application logic.
- JSPs are used for presentation of content.

- Tag libraries and JSP EL are used instead of embedding Java code inside JSPs.
- MVC frameworks such as Struts are used to aid in the separation of presentation and application logic.

The next section briefly looks at the structure of a typical Web application, and how it is built and distributed. Chapter 7 discusses this structure in far more detail.

Building and Distributing Web Applications

The set of all the servlets, JSPs, and other files that are logically related constitutes a *Web application*. Such a Web application is packaged as a *WAR* (for Web Application Archive) file, with a .war extension.

The Servlet specification defines a standard directory hierarchy for the files inside a WAR. It is described in the following table.

Relative Path	Description
`/`	Web application root: All files that are publicly accessible are placed in this directory. Examples include HTML, JSP, and GIF files.
`/WEB-INF`	All files in this directory and its subdirectories are not publicly accessible. A single file, `web.xml`, called the *deployment descriptor*, contains configuration options for the Web application. The various options for the deployment descriptor are defined by the Servlet API.
`/WEB-INF/classes`	All of the Web application's class files are placed here.
`/WEB-INF/lib`	Class files can be archived into JAR files and placed in this directory.

A WAR file is built using the Java `jar` (Java archive) command, either directly as shown here, or using a build tool, such as Ant. Ant is covered in greater detail in Appendix B.

```
jar cvf application.war application/
```

Here, `application.war` is the WAR file to be created and `application/` is the directory containing the Web application code and configuration in the format specified in the previous table.

All Servlet containers are required to use this directory hierarchy for WAR files. What's more, because the location and features of the deployment descriptor (the `web.xml` file mentioned previously) are set by the specification, Web applications need to be configured only once and they are compatible with any Servlet container. The deployment descriptor defines options such as the order in which servlets are loaded by a Servlet container, parameters that can be passed to the servlets on startup, which URL patterns map to which servlets, security restrictions, and so on. Chapter 7 provides a full description of the deployment descriptor.

This means that developers need expend effort creating a Web application only once. Thus, distributing and deploying Web applications is remarkably simple, even if you switch Servlet containers.

Server administrators can deploy the WAR file in the Servlet container (such as Tomcat), and the Servlet container takes care of the rest. Deploying WAR files is covered in great detail in Chapter 8.

Summary

To conclude this chapter, let's review some of the key points that have been discussed:

- The basic Internet protocols, HTTP and HTML, did not offer ways to generate dynamic content or maintain user state over multiple requests. CGI scripts and the various server-side Java technologies were invented for this purpose.
- CGI scripts have limitations, including poor scalability and performance. Servlets and JSPs overcome these limitations.
- Servlets are a server-side Java technology that is used for generating dynamic content and implementing application processing logic.
- JSPs are a better choice for generating dynamic content than servlets are, as HTML statements don't need to be embedded inside Java code.
- JSPs are compiled into servlets, which are then kept in memory or on the file system indefinitely, until either the memory is required back or the server is restarted. A difference between servlets and JSPs is that servlets are provided in compiled form and JSPs are often not (although pre-compilation is possible).
- Replacing Java code scriptlets with JSP tag libraries improves maintainability of the Web application. Each tag has a corresponding Java class that contains the code that would otherwise appear on the page.
- The set of all the servlets, JSPs, and other files that are logically related constitutes a *Web application*. Web applications are packaged into WAR (Web application archive) files, and the Servlet specifications define a standard directory structure for them.
- Servlets are ideal for encapsulating the logic of the application, while being somewhat poor at visual representation; conversely, JSP is designed for displaying visual material. Therefore, the combination of the two can provide a balance and cover the needs of most sites. The architecture that aids this separation between logic and presentation is known as Model 2 architecture or Model View Controller (MVC) architecture.

In Chapter 3, you learn how to install Tomcat.

3

Tomcat Installation

The previous two chapters provided a background to Tomcat: a brief history, Tomcat licensing details, an overview of where Servlet containers such as Tomcat fit in the enterprise Java (Java EE) stack, and an introduction to Web application technologies (servlets, JSPs, tag libraries, and so on). With that out of the way, you can now move on to installing Tomcat. In later chapters, you learn the details of configuring it.

This chapter covers the following aspects of Tomcat installation:

- Installing the Java Virtual Machine (JVM)
- Installing Tomcat on both Windows and Linux
- Understanding the Tomcat installation directory structure
- Troubleshooting typical problems encountered while installing Tomcat

If you have installed earlier Tomcat versions (Tomcat 5.5 and before), you might be tempted to skip this chapter and move ahead. Indeed, Tomcat installation is reasonably straightforward. Check if you have JVM installed and in `PATH`, grab the Tomcat binary from the Apache Web site, unzip or run the installable, and you are good to go.

However, Tomcat 6 introduces some twists — a changed directory structure, and the requirement of Java SE 5 JVM. For that reason alone, we recommend that you read this chapter before going forward.

Installing the Java Virtual Machine

Tomcat, like any Java-based application, requires a Java Virtual Machine (JVM) to function. Sun Microsystems distributes a free JVM for Windows, Linux, and Solaris. Other third-party vendors and open-source groups make JVMs for other platforms — some for free, others commercially.

Before installing Tomcat 6, you need to ensure that Java 5 JVM is installed on your system. To check the JVM version, type the `java -version` command as shown.

```
C:\> java -version
java version "1.5.0_03"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_03-b07)
Java HotSpot(TM) Client VM (build 1.5.0_03-b07, mixed mode)
```

A version string of the form *1.5.x* or higher indicates that you have the right version of JVM installed for Tomcat. This command assumes that `<JAVA_HOME>/bin` is in your `PATH`.

The following sections show you how to install the JVM; if you already have this installed you can skip ahead to the section "Installing Tomcat."

Tomcat versions preceding Tomcat 5.5 required the installation of the Java Development Kit (JDK) and not just the Java Runtime Environment (JRE). The JDK is meant for developers to be able to compile Java programs, and has the development tools such as the Java compiler (`javac`), debugger, and development libraries. The Java compiler was used by earlier versions of Tomcat to compile JSPs at runtime, and hence just installing the JRE was not enough.

Tomcat versions 5.5 and 6 have a Java compiler (the *Eclipse JDT Java compiler*) packaged along with them. This is used to compile JSP pages; and hence you can run Tomcat 5.5 and 6 with a Java 5 JRE only. Both the JRE as well as the JDK contain the Java runtime (i.e., the JVM).

Installing the JVM on Windows

In the Windows environment, the installer is an executable with easy-to-follow steps. First, download the latest JDK or JRE from Sun's Java Web site:

```
http://java.sun.com
```

> **Tomcat 6 requires Java SE version 5.0 or later to run.**

Double-click the downloaded file and you will soon have the JDK/JRE installed. The folder in which you have chosen to install the JDK/JRE is known as your Java Home folder. It contains several subfolders, but the only one of interest here is the `bin` directory in which the various executables are stored (including the JVM, and for the JDK, the compiler, the debugger, and a packaging utility).

The next step of the installation is to add the Java Home folder as an environment variable named `JAVA_HOME` so that Windows can find it when it is invoked. The `bin` subdirectory of the Java Home folder should also be added to the `PATH` environment variable.

To do this on Windows XP, select Start➪Control Panel and choose the System option. Now choose the Advanced tab and select the Environment Variables button. If desired, you can add the variable settings to the specific user that you are operating as, so that it will exist only when you are logged in as that user. Alternatively, you could add the settings for the entire system.

To add the JAVA_HOME environment variable for the entire system, select the New button in the lower half of the window, as shown in Figure 3-1.

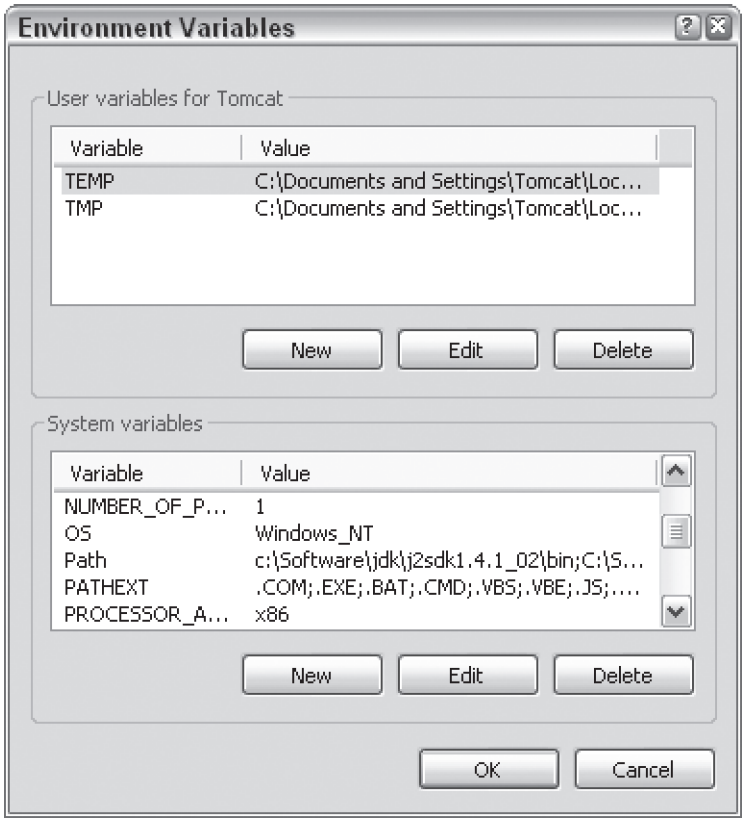

Figure 3-1: Selecting the New button

Now enter the variable name and value, as shown in Figure 3-2.

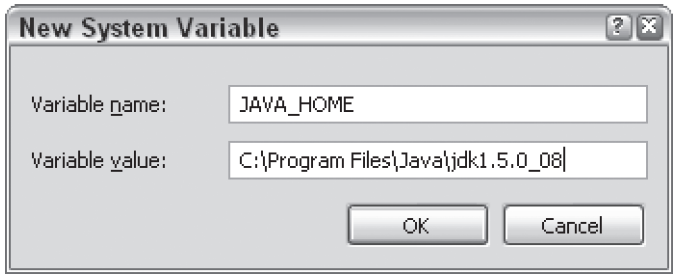

Figure 3-2: The JAVA_HOME information

This information may vary depending on the specific version of the JVM you have installed and the location of your installation. Next, modify the PATH variable to include %JAVA_HOME%\bin, making sure that it is the first entry in PATH, as shown in Figure 3-3.

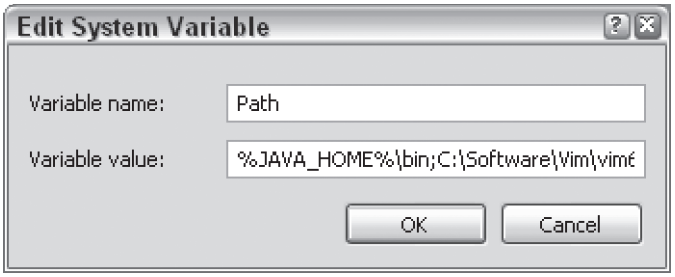

Figure 3-3: Modifying the Windows PATH

This will make the Java executables available to the command prompt. To test the installation, open an instance of the command prompt (Start➪Programs➪Accessories➪Command Prompt) and enter **javac** in the command window.

This should bring up a standard usage message such as the following (cropped short here):

```
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
```

Installing the JVM on Linux

For a Linux installation, first download a suitable distribution. Sun's JVM can be downloaded from `http://java.sun.com`. A good alternative to Sun's JVM on Linux is IBM's implementation (`http://www.ibm.com/developerworks/java/jdk/`). The following section, however, uses Sun's JVM.

> **Tomcat 6 requires Java SE version 5.0 or greater to run.**

The official supported platform is Red Hat Linux, but Sun's JDK and JRE can be adapted to work with other distributions without too much trouble.

The following sections describe the two types of download: a tar/gzip version and an RPM package for systems supporting RPMs. The only difference between the two versions is that RPM would install the JVM in a location consistent with RedHat conventions (under `/usr/java`), whereas the tar/gzip version gives more control to users on where to put the JVM. Also, installing with RPM allows administrators to use the RPM package management system to query for which versions of Java are installed, instead of hunting around the file system for them.

Installing the JVM from the tar.gz File

For the tar/gzip version, the installation process is as follows: Once the archive has been downloaded, extract its contents — a single self-extracting binary file. Installing the JDK for all users is demonstrated in these instructions. To do this, you must log in as root, or execute the command with root privileges using the `sudo` command. Begin by moving the file into the directory in which you would like to install the JDK or JRE.

If you are installing the JDK for a specific user, then you must install it into the user's home directory. Alternatively, if you wish to install the JDK for all users, then the default location is `/usr/java/jdk-[version number]`, where *version number* is the version number of the JDK being installed.

Now add execute permissions for the file as follows:

```
# chmod o+x jdk-1_5_0_06-linux-i586.bin
```

Run the file using the following line:

```
# ./jdk-1_5_0_06-linux-i586.bin
```

You will be presented with a license agreement before installation commences. Once installation has finished, you should add the environment variable `$JAVA_HOME` to your system, with the location of the JDK. For example, if you installed it in `/usr/java/jdk-1_5_0_06-linux-i586`, you should give it this value. This value can be added to the `~/.bashrc` file for personal use or to `/etc/profile` for all users. Alternatively, `/etc/profile` runs any shell scripts in `/etc/profile.d`, so the following lines can be added to a script (here named `tomcat.sh`) in that directory. (Change the details of the Java directory as appropriate.)

```
JAVA_HOME=/usr/java/jdk-1_5_0_06-linux-i586/
export JAVA_HOME
PATH=$JAVA_HOME/bin:$PATH
export PATH
```

Note that you may have to log out and log in again for `/etc/profile` or `tomcat.sh` to be read by your system. You should also allow execute permissions for the `$JAVA_HOME/bin` folder for all users or for yourself as owner as appropriate.

To test the installation, type **javac.**

This should provide the following output (cropped for the sake of brevity):

```
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
```

Installing the JVM Using the RPM Installer

To install the JVM using the RPM, you must first download the file. The format is as follows:

```
j2sdk-[version number]-linux-i586-rpm.bin
```

On executing this file, you are presented with the license terms for Apache. After execution, an RPM with the same name, but with the trailing `.bin` removed, is automatically uncompressed. To run the RPM, you must sign in as root, or execute the command with root privileges using the `sudo` command. Set execute permissions for the file as shown and then run it:

```
# chmod o+x jdk-1_5_0_06-linux-i586-rpm.bin
# ./jdk-1_5_0_06-linux-i586-rpm.bin
```

The script will display a binary license agreement, which you will be asked to agree to before installation can proceed. Once you have agreed to the license, the install script will create the RPM file in the current

directory. To install the RPM, type the following:

```
# rpm -iv jdk-1_5_0_06-linux-i586-rpm.bin
```

This will install the Java 2 SDK at `/usr/java/jdk1.5.0_06`. You should now follow the previous instructions to modify various environment settings. You should also test the installation as described earlier.

Installing Tomcat

A convention followed in the rest of the book is to refer to the Tomcat installation directory as TOMCAT_HOME or CATALINA_HOME.

For each of the following steps (for Windows, Linux, and Unix systems), you can download the distributions from the same folder on the Tomcat Web site. Navigate to the following URL:

```
http://tomcat.apache.org/
```

Click the download binaries link (which, at the time of this writing, is on the left side) and choose the most recent (stable) Tomcat 6 version. The download page randomly selects a mirror site for you to download from, and this can be changed using the drop down menu on the page.

> **Caveat: One of the mirror sites for downloading Tomcat, as of this writing, is `mirrors.playboy.com`, which is blocked (and flagged!) by a lot of corporate proxy servers — so make a note of what proxy site you are assigned to by default.**

Deciding Which Distribution to Install

The Tomcat Web site lists four different distributions for every Tomcat 6 version: core, deployer, embedded and the admin webapp. What are these, and which should you download?

The distributions are:

- **Core:** The base version of Tomcat, and is available either as a tar.gz file (`apache-tomcat-`*`version`*`.tar.gz`), a ZIP file (`apache-tomcat-`*`version`*`.zip`), or a Windows executable installer (`apache-tomcat-`*`version`*`.exe`). Installing from either of these is covered later in the chapter.
- **Deployer:** This is the standalone Tomcat Web application deployer.

Admin webapp: Currently not available for Tomcat 6. In earlier versions of Tomcat, it was a Web application that was used to administer Tomcat and was packaged separately for security reasons.

For a typical server deployment — i.e., installing Tomcat on a server machine, and deploying Web applications on it — you need to download the *core* distribution.

If you have an existing Tomcat deployment, say on a production server, and you have to deploy new Web applications to it, then you should get the *deployer* distribution. This client-side application helps you compile Web applications, validate them, generate WAR (Web archive) files, and deploy them to a Tomcat server. This is covered in Chapter 8.

Earlier versions of Tomcat also came with an embedded distribution: If you don't know what this is, then chances are that you don't need this. An embedded Tomcat instance is required when an application needs to have a Servlet container "embedded" inside it. Chapter 18 covers how Tomcat can be run in this mode.

Verifying the Downloaded File

After you download a Tomcat distribution, you must validate it. This means that you should check the binary against the published checksum and signature. This is very important because you have downloaded the Tomcat distribution over an unsecured connection, and also from one of the many Apache mirror sites. Each distribution has a *PGP signature* and *MD5 checksum* listed along with the download link, and these are used to verify the integrity of the downloaded software.

The MD5 checksum is a unique, 128-bit "digest" for the downloaded file. Its acts like a fingerprint for the file, and the MD5 algorithm ensures that no two files will have the same checksum. You can verify the checksum by computing it for the file downloaded from the mirror site, and verifying it against the checksum listed on the Tomcat Web site. The following command shows you how to compute this checksum on Linux:

```
$ md5sum apache-tomcat-6-x-y.zip
f805b44ccdebc7c276b801d3d9e42ac2    apache-tomcat-6-x-y.zip
```

For Windows, the md5sum program can be downloaded from `www.fourmilab.ch/md5/`, `www.pctools.net/win32/`, `www.slavasoft.com/fsum/`, or as a part of Cygwin (`www.cygwin.com/`), the Unix-like environment for Windows.

Pretty Good Privacy (PGP) is a cryptographically stronger mechanism to verify the integrity of the downloaded file. To verify the download using PGP, first get the `KEYS` file from the Tomcat download page. Currently, this `KEYS` file is located at `apache.org/dist/tomcat/tomcat-6/KEYS`; however, check the URL listed on the download page.

After this, use one of the following commands to verify the PGP signature, depending on which implementation (PGP, GPG) is installed:

For PGP:

```
$ pgp -ka KEYS
$ pgp apache-tomcat-6-x-y.zip
```

For GPG:

```
$ gpg --import KEYS
$ gpg --verify apache-tomcat-6-x-y.zip
```

In the preceding commands, the first command (`pgp -ka, gpg --import`) imports the PGP/GPG keys of the Tomcat developers from the `KEYS` file that was downloaded from the Apache Web site. This is then used to validate the integrity of the downloaded software.

GPG, also called GnuPG or Gnu Privacy Guard, is a free replacement of PGP. Binaries for a number of platforms are available from `www.pgpi.org/products/pgp/versions/freeware/`, while the commercial version is available from `www.pgp.com`.

Tomcat Windows Installer

The Tomcat download page contains many different links for Tomcat 6.*x*. The one you want has an extension of .exe. Save this file in a convenient location on your machine, and double-click it to begin installation.

Once you've agreed to the Apache license, the installer presents you with a Choose Components screen, as shown in Figure 3-4. You should probably select the Full option, which will install all of the Tomcat components. Some of the components deserve some discussion, however.

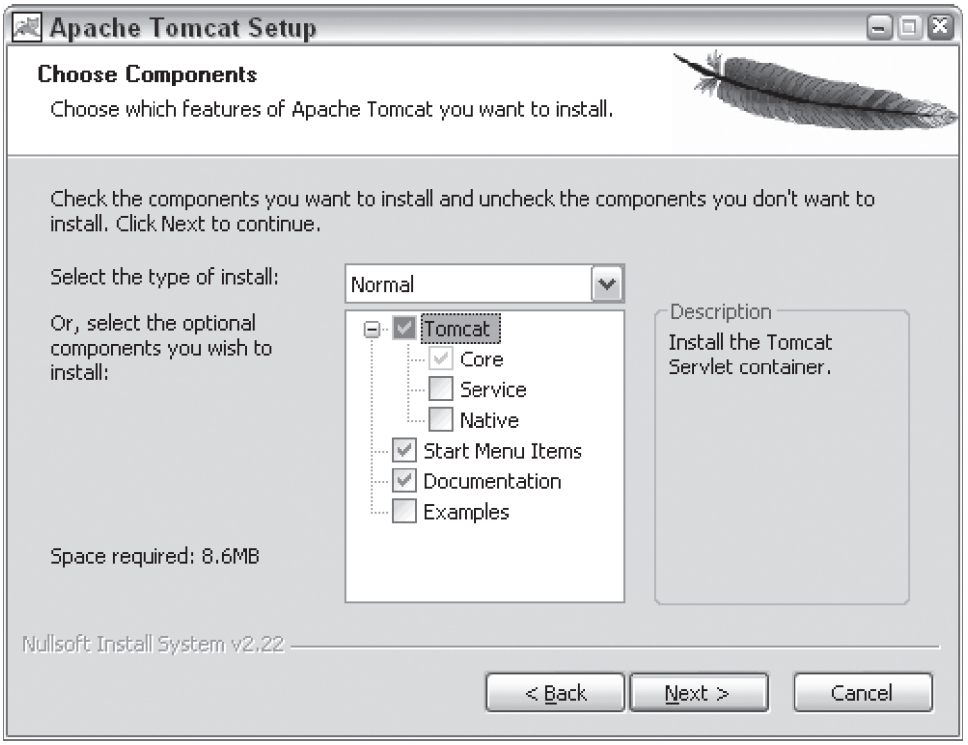

Figure 3-4: The Tomcat setup's Choose Components screen

The Service Component

One component you may not wish to install is the Service component (a subcomponent of Tomcat; if you can't see it, click the plus symbol next to Tomcat). The Service component enables you to start, stop, and restart Tomcat in the same way as any other Windows service, and this option is recommended if you are accustomed to managing your system services in this way. The chief advantage of a service is that it will automatically start Tomcat for you when your system starts, and it will do so without displaying any command prompts or other open windows.

A service is clearly the better option for production servers, but may not be what you want for development; starting and stopping a service repeatedly can be a pain.

Finishing the Installation

Once you've chosen the components you wish to install, click the Next button. You will be prompted to choose a directory into which Tomcat should be installed. While the default directory is `C:\Program Files\Apache Software Foundation\Tomcat 6.x`, you should consider installing Tomcat into a path that does not contain spaces in the name, such as `c:\java\tomcat-6.x`. Once you've reviewed the destination folder, click Next.

The next screen requests the Tomcat port and an administrator login. Leave the port value as 8080, but choose a unique username for the administrator login, and select a hard-to-guess password. When you are done, click Next.

The final screen will ask for the location of the JDK you installed earlier. Enter it if it was not automatically found. Then, click Install.

Setting Environment Variables

While not strictly required when Tomcat's Windows installer is used, it is a good idea to add an environment variable that points to the directory in which you installed Tomcat. The environment variable is named `CATALINA_HOME`. To add the environment variable, navigate to your Control Panel and choose System. Now choose the Advanced tab and select the Environment Variables button. Select the New button in the system variables (lower half) section and enter `CATALINA_HOME` as the variable name and the path to your Tomcat installation (for example, `c:\java\tomcat-6.0`).

Testing the Installation

To test the installation, you must first start the server. You can start Tomcat manually or, if you installed Tomcat as a service, you can start the service.

Starting the Server Manually

You can start the server manually by selecting Start➪Programs➪Apache Tomcat 6 and choosing Start Tomcat. A new command-prompt window will appear, demonstrating that the server is running.

Alternatively, you can start Tomcat by opening a command-prompt window, navigating to `<TOMCAT_HOME>\bin`, and typing `tomcat6.exe`, as shown in Figure 3-5. The standard Tomcat startup script (`startup.bat`) is not present when installing from the installer.

If Tomcat does not start up, you can find some troubleshooting tips at the end of this chapter. You may also get error messages if your `%JAVA_HOME%` variable is not defined, and if the `%JAVA_HOME%\bin` directory within the JDK is not in the `PATH`. If this is the case, you will get an error message such as the following:

```
'java' is not recognized as an internal or external command, operable program or
batch file.
```

Refer to the instructions in the section "Installing the Java Virtual Machine," earlier in this chapter, if this is the case.

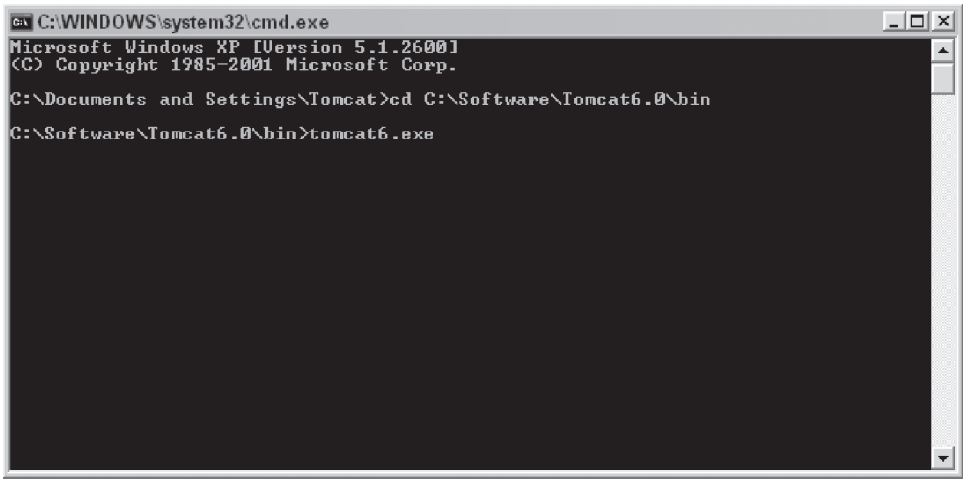

Figure 3-5: Starting Tomcat from the command line

To shut down Tomcat, use the Shutdown shortcut (Start➪Programs➪Apache Tomcat 6➪Stop Tomcat) or type `shutdown` into the command prompt from Tomcat's `bin` directory.

Starting the Server as a Service

If you wish to start the server as a service (and assuming you chose this option when installing Tomcat), you will need to start up the service. This is done by double-clicking Administrative Tools in the Control Panel. In Administrative Tools, you should select Services. In the window that opens, you should find an entry for Tomcat, as shown in Figure 3-6.

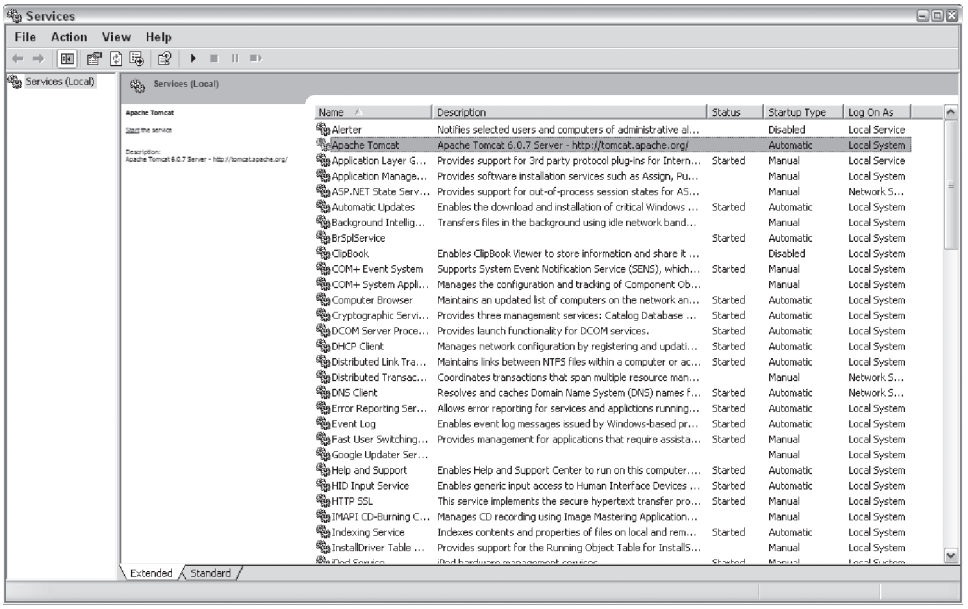

Figure 3-6: The Apache Tomcat service

To start the server, right-click the Tomcat entry and choose Start. No window will appear because the server is running as a service in the background. Once the server is started, the options for restarting and stopping the server will also be enabled.

Changing Service Options

Looking at Figure 3-6, you can see that the Startup Type is set to Automatic, which means that restarting the computer also starts an instance of Tomcat automatically. From now on, every time Windows is started, Tomcat will automatically start up at boot time and will be available from then on.

You can further customize the service by choosing the Properties option from the context menu. This enables the startup type to be changed to `Manual`, or for the service to be disabled entirely. It also enables you to choose to automatically restart the service should it crash. This last option is especially useful because not only does it allow you to reboot the computer, but it also enables you to run a script should the server fail.

You can also perform different actions depending on how many times the service has failed (by choosing the Recovery tab), so you can initially request a reboot of the service, then a reboot of the machine, after which any subsequent failures will cause a script to run that perhaps alerts you of the failure.

If you wish to set the recovery options, right-click the Tomcat service entry in the list and choose Properties. In the window that opens, choose Recovery, and you should be presented with the options shown in Figure 3-7.

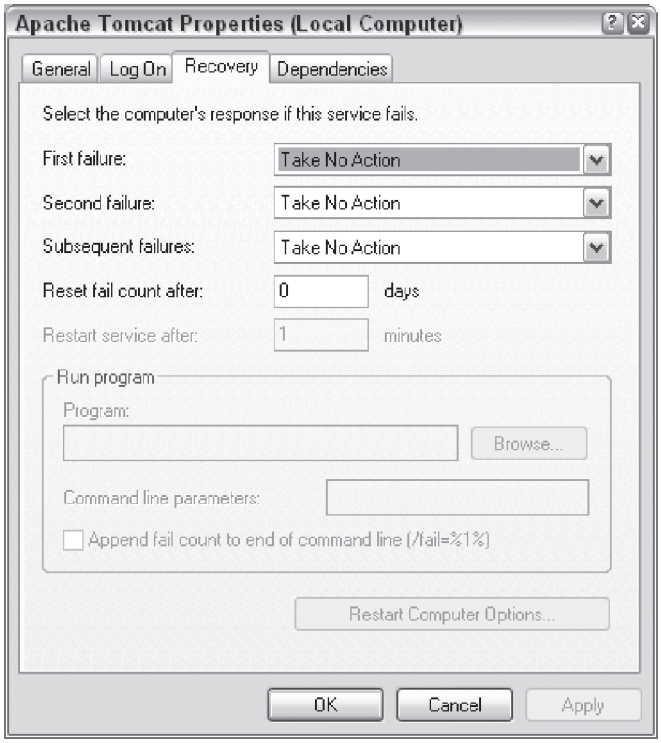

Figure 3-7: The Recovery options

As you can see, the default is for no action to be taken. As you desire, you can configure the service to be restarted on failure, and/or run programs when a failure occurs.

Viewing the Default Installation

Tomcat, like most servers, comes with a default home page that can be used to confirm that the installation is working. Enter the following address in a browser:

```
http://localhost:8080/
```

The page shown in Figure 3-8 should appear.

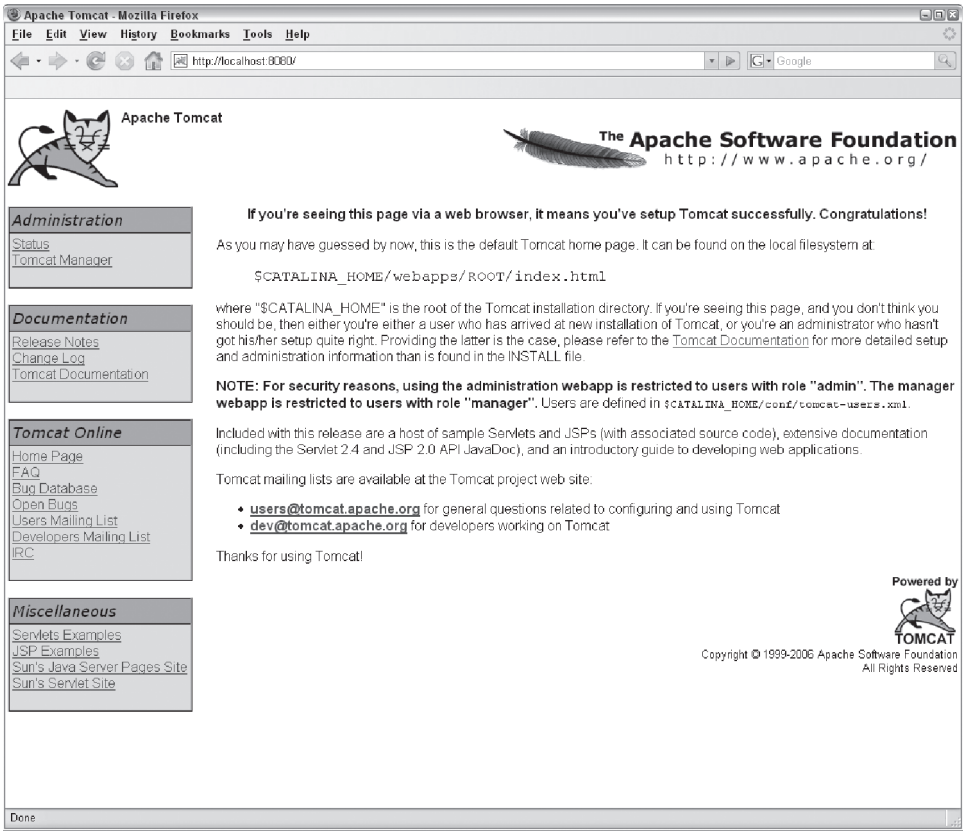
Apache Tomcat - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/
Apache Tomcat
The Apache Software Foundation
http://www.apache.org/
Administration
Status
Tomcat Manager
Documentation
Release Notes
Change Log
Tomcat Documentation
Tomcat Online
Home Page
FAQ
Bug Database
Open Bugs
Users Mailing List
Developers Mailing List
IRC
Miscellaneous
Servlets Examples
JSP Examples
Sun's Java Server Pages Site
Sun's Servlet Site
If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!
As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:
$CATALINA_HOME/webapps/ROOT/index.html
where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then either you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.
NOTE: For security reasons, using the administration webapp is restricted to users with role "admin". The manager webapp is restricted to users with role "manager". Users are defined in $CATALINA_HOME/conf/tomcat-users.xml.
Included with this release are a host of sample Servlets and JSPs (with associated source code), extensive documentation (including the Servlet 2.4 and JSP 2.0 API JavaDoc), and an introductory guide to developing web applications.
Tomcat mailing lists are available at the Tomcat project web site:
users@tomcat.apache.org for general questions related to configuring and using Tomcat
dev@tomcat.apache.org for developers working on Tomcat
Thanks for using Tomcat!
Powered by
TOMCAT
Copyright © 1999-2006 Apache Software Foundation
All Rights Reserved
Done

Figure 3-8: The default Tomcat home page

Assigning Port Numbers

The default installation requires you to include the port number assignation (for example, `:8080`) in the server address. Ports are logical addresses in a computer that enable multiple communications with the server and the channeling of different protocols. For example, SMTP is addressed to port 25, SSH is

addressed to port 22, Telnet to 23, and so on. Browsers automatically point at port 80 if no port is specified (443 for HTTPS); hence, the use of ports is not immediately visible to the average user.

Because the majority of server hardware already includes a standard Web server installation (usually Apache for Linux, and IIS for Windows), Tomcat does not attempt to connect to the standard HTTP traffic port (which is 80 by default), but rather to port 8080.

The configuration file that specifies the port number is called `server.xml` and can be found in the installation folder of Tomcat in the `%CATALINA_HOME%\conf` directory. It's just a text file, and somewhere within it you should find an entry similar to the following:

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 -->
<Connector acceptCount="100" connectionTimeout="20000" debug="0"
    disableUploadTimeout="true" enableLookups="false" maxSpareThreads="75"
    maxThreads="150" minSpareThreads="25" port="8080" redirectPort="8443"/>
```

You can find this entry by searching for the string `port="8080"`. Changing this to another number will change the Tomcat port number. Changing it to 80 enables you to connect to Tomcat using the following URL without the trailing colon and port number:

```
http://localhost/
```

If you have any problems, refer to the "Troubleshooting and Tips" section at the end of this chapter.

Conversely, if all has gone well, you are now the proud owner of your own Tomcat instance. Before you are finished, you should check Tomcat's capability to serve JSP pages and servlets.

To do this, choose the JSP Examples link from the left-hand menu and select some of the examples to run. Confirm that they all run as they are supposed to without error messages. Do the same for the Servlet Examples link to test this functionality.

Installing Tomcat on Windows Using the ZIP File

Installing Tomcat using the ZIP file is not much different from the process described earlier. The ZIP file is provided for those who prefer to manually install Tomcat.

To install Tomcat using the ZIP file, simply unpack the contents of the file to your directory of choice, such as `c:\java\tomcat-6.0`.

Now add the `%CATALINA_HOME%` environment variable as shown in the preceding directions. To check your installation, you need to follow slightly different instructions than before. Because the shortcuts for the server are not created automatically, you need to call a couple of batch files provided in the `%CATALINA_HOME%\bin` directory for this purpose.

To start the server, type the following at the command prompt:

```
> cd %CATALINA_HOME%\bin
> startup.bat
```

As with the preceding installation method, a new window will open, indicating that the server has started. To shut down Tomcat, type **shutdown.**

Installing Tomcat on Linux

Installing Tomcat on Linux or Unix is easy. Download the tar/gzip file of the latest Tomcat 6.*x* binary release from the following URL:

```
http://tomcat.apache.org/
```

Extract the downloaded file onto your hard drive to a path such as `/usr/java/jakarta-tomcat-6`. Note that you should use the GNU version of the `tar` utility to ensure that long file names are handled properly.

You should now export the `$CATALINA_HOME` variable, using the following command (in `bash`):

```
# CATALINA_HOME=/usr/java/ tomcat-6
# export CATALINA_HOME
```

The Tomcat 6 release notes also recommend that if you are on GLIBC 2.2 / Linux 2.4, you should define an additional environment variable as shown:

```
# export LD_ASSUME_KERNEL=2.2.5
```

Also, if you are on Redhat Linux 9.0, you should use the following setting:

```
export LD_ASSUME_KERNEL=2.4.1
```

These settings avoid known stability problems with Tomcat. You can check your `glibc` (i.e., the GNU C Library) version on Redhat using the `rpm -q glibc` command.

All these commands can be added to your `~/.bashrc` or `/etc/profile` as you did for the JDK installation, or you can create a shell file, `tomcat.sh`, and place it in `/etc/profile.d`. It will be run automatically by `/etc/profile` at boot time to make the variable available to all users.

You can now start Tomcat by running the following shell command:

```
# $CATALINA_HOME/bin/startup.sh
```

Another approach to installing Tomcat on Linux is to use the package manager tool specific to the Linux distribution, such as `emerge` on Gentoo Linux or `apt-get` on Debian to download and install. This is often a very convenient and familiar approach for administrators, and gives them a stable version of Tomcat to work with.

However it should be noted that these tools may place Tomcat configuration files in non-standard places. The following table summarizes the differences between the standard Tomcat directory structure, and the one created by installing Tomcat using Gentoo's `emerge` command. In the rest of the book, we assume a standard Tomcat directory structure is in place. If you are using a Tomcat installed with the Gentoo conventions, use the table to map the directory paths appropriately.

Path on Gentoo	Standard Tomcat Path
/usr/share/tomcat-x.y/bin	<TOMCAT_HOME>/bin
/usr/share/tomcat-x.y/common	<TOMCAT_HOME>/common for Tomcat 5.5 and before; not present in Tomcat 6.
/etc/tomcat-x.y	<TOMCAT_HOME>/conf
/var/log/tomcat-x.y	<TOMCAT_HOME>/logs
/usr/share/tomcat-x.y/server	<TOMCAT_HOME>/server for Tomcat 5.5 and before; not present in Tomcat 6.
/var/lib/tomcat-x.y/shared	<TOMCAT_HOME>/shared for Tomcat 5.5 and before; not present in Tomcat 6.
/usr/share/tomcat-x.y/lib	<TOMCAT_HOME>/lib for Tomcat 6. Not present in earlier versions, i.e., Tomcat 5.5 and before.
/var/tmp/tomcat-x.y	<TOMCAT_HOME>/temp
/var/lib/tomcat-x.y/webapps	<TOMCAT_HOME>/webapps
/var/run/tomcat-x.y	<TOMCAT_HOME>/work

Tomcat installations on Gentoo Linux, a popular Linux distribution, have a different directory structure than standard Tomcat installations.

Instead of using the Gentoo `emerge` command to install, administrators can install the standard Tomcat distribution as on other Linux distributions.

Viewing the Default Installation

To confirm that Tomcat is running, point your browser to the following URL:

```
http://localhost:8080/
```

Choose the JSP Examples link from the menu on the left-hand side and select some of the examples to run. Confirm that they run without error messages. Do the same for the Servlet Examples to test their functionality.

Modifying Port Numbers

Tomcat uses port 8080 by default. Because the majority of server hardware already includes a standard Web server installation, usually Apache, Tomcat does not attempt to connect to the standard HTTP traffic port, 80, by default.

The configuration file that specifies the port number is called `server.xml` and can be found in the `$CATALINA_HOME/conf` directory. Somewhere within it you should find the following entry:

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 -->
<Connector acceptCount="100" connectionTimeout="20000" debug="0"
    disableUploadTimeout="true" enableLookups="false" maxSpareThreads="75"
    maxThreads="150" minSpareThreads="25" port="8080" redirectPort="8443"/>
```

You can find this entry by searching for the string `port="8080"`. Changing this to another number (higher than 1024 in Linux) will change the Tomcat port number. Changing it to 80 will enable you to connect to Tomcat using the URL `http://localhost`, providing that the Tomcat server is started with root permissions. The reason for running Tomcat as root is that on Unix (and Linux) systems, non-root processes cannot bind to ports lower than 1024.

However, running Tomcat as the super user is a *very bad idea*, because any badly written (from a security perspective) Web application can compromise your system. If you want Tomcat to listen on port 80, there are a number of other, more secure alternatives to running it as root:

- Run Apache Web server on port 80 and configure Apache to send requests to Tomcat. (Chapter 11 discusses this in more detail.) However, this solution just pushes the problem over to Apache because now Apache has to deal with how to bind to port 80 without running as root.
- Run Tomcat on a non-privileged port, such as 8080 and use a "port redirector" such as `rinetd` to redirect messages coming to port 80 to port 8080.
- Run Tomcat on a non-privileged port, such as 8080 and configure the firewall to redirect external requests to port 80 to the internal port 8080. Because production Web sites almost always have a firewall in deployment, this is often the most commonly used procedure.

If you have any problems installing, refer to the "Troubleshooting and Tips" section at the end of this chapter.

Building Tomcat from Source

While downloading Tomcat from the Apache Web site, you can see that both source codes as well as binary versions of Tomcat are available. This section explains how (and why) you would build and install Tomcat from source.

Do You Need to Build Tomcat from the Source Code?

The short answer is no.

If you are an administrator trying to install a production-ready release of Tomcat, you almost never would need to download and build a source release of Tomcat. If this is the case, feel free to skip this section and move on to the next major section in the chapter, "The Tomcat Installation Directory."

You would want to download a source release if you are developer who wants to experiment with Tomcat, perhaps even extend it. Or, if you have a bug in your particular Tomcat version, and you want to download the fix for it and patch it in, without waiting for the next official release. In the second case, you would be downloading the patch from the Tomcat Subversion repository.

Downloading the Source Release

The simplest way to get a source release is to download a labeled release from the Tomcat download site (`http://tomcat.apache.org`). The usual caveat of verifying the release (MD5 checksum, PGP signature) applies in this case, too. These releases are usually labeled as `apache-tomcat-6.x.y.tar.gz` or `apache-tomcat.6.x.y.zip`.

Subversion Repository

Tomcat, like a lot of other Apache projects, has migrated from using CVS as its version control system to Subversion. *Subversion* is designed to be a replacement of CVS, and overcomes a lot of the deficiencies that CVS had. More information on Subversion, including downloads of clients and servers, is available at `http://subversion.tigris.org/`. A popular Windows client for Subversion is Tortoise SVN (`http://tortoisesvn.tigris.org/`), which integrates into Windows Explorer. The commands shown here, however, are for a Linux-based, command-line client.

```
$ svn co http://svn.apache.org/repos/asf/tomcat container
```

In the preceding command, the URL (`http://svn.apache.org/repos/asf/tomcat`) specifies the root of the Tomcat Subversion repository, and `container` is the module being checked out. If you are familiar with the CVS version control system, you can think of the Subversion repository root URL as analogous to `CVSROOT`. However you can browse the root URL using any Web browser, and view the code without downloading it. You can do neat stuff like this because the Subversion server makes use of the Apache HTTP server, and a *WebDAV* (`www.webdav.org/`) module. Subversion has a standalone server, too, just as CVS does, but the Apache/WebDAV combination is more commonly used.

The container module contains the core Tomcat code; the list of all other modules is at `http://tomcat.apache.org/svn.html`.

Building a Source Release

Once you have downloaded the source code, building Tomcat is a simple matter of running the Ant build script present in the top-level directory. Ant is a build utility for Java programs. If you are not familiar with using it, refer to Appendix B.

The commands that follow assume that both Java (`<JAVA_HOME>/bin`) and Ant (`<ANT_HOME>/bin`) are defined in the system `PATH`.

```
$ cd apache-tomcat-6-x-y
$ ant
```

The Ant build script (`build.xml`) specifies which third-party jar files are to be downloaded, and as the script executes, you will see the required components either downloaded as required, or checked out from CVS repositories — not all Apache projects have been migrated to Subversion at the time of this writing. Naturally, the machine that the build is run on requires Internet access, as well as a command-line CVS client. If not, you will see errors during the build process. If the machine that you are running the build on connects to the Internet through a proxy server, you have to specify this in a `build.properties` file. Copy the sample file from the `apache-tomcat-6` directory to the parent directory, rename it to `build.properties`, and edit the following lines as required:

```
# ----- Proxy setup -----
proxy.host=proxy.domain
proxy.port=8080
proxy.use=on
```

The Tomcat Installation Directory

After your install Tomcat, you should see the directory structure shown in Figure 3-9.

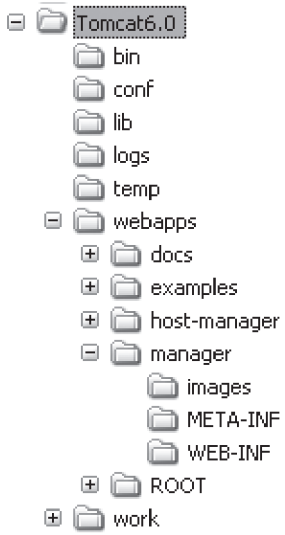

Figure 3-9: Tomcat 6 directory structure

As you can see, there are differences between the Tomcat 6 directory structure and that of older Tomcat versions:

- Tomcat 5.5 and earlier Tomcat versions have `shared`, `common`, and `server` directories under the Tomcat install directory. The `<TOMCAT_HOME>/shared` directory is to be used by Web applications that want to share classes and JAR files with each other; the `<TOMCAT_HOME>/common` directory contains class files and JAR files that were visible to both the Tomcat server, as well as all deployed Web applications. The `<TOMCAT_HOME>/server` directory, on the other hand, contains classes and JAR files visible only to Tomcat.
- Tomcat 6 does not have these directories. Instead, it has a `lib` directory (`<TOMCAT_HOME>/lib`), where all the JAR files used by Tomcat go. This is visible to all Web applications, too; however, Web application–specific JAR files should not be placed here.

This directory structure is explained in greater detail in the next chapter, but briefly, the different directories are:

- `bin`: The `bin` directory contains the shell scripts and batch files for starting Tomcat in various modes. It also includes a pre-compiler for JSP pages that can improve startup time and first-time response (the time it takes for the server to respond to a request for a JSP page that has not been previously compiled). Compilation occurs only once, but it can frustrate the first visitor to a site after the server is restarted because of the long response time.
- `conf`: The `conf` directory contains the configuration files for Tomcat. These include general server configuration files, a default user list for file-based authentication and security for

Web applications, and a global configuration file. Later chapters discuss these files in greater detail.

- `logs`: The `<TOMCAT_HOME>/logs` directory contains the server logs.
- `lib`: The `<TOMCAT_HOME>/lib` directory contains all the JAR files used by Tomcat.
- `webapps`: This directory contains all the Web applications provided with Tomcat by default. This includes the Tomcat manager application that is discussed in Chapter 8. User-developed Web applications will also be deployed to this directory. The structure of the `webapps` directory is discussed in greater detail in Chapter 7.
- `work`: The `work` directory contains temporary files, precompiled JSP pages, and other intermediate files.

Installing APR

An optional component with Tomcat is APR, or the Apache Portable Runtime. This is a library that was originally developed as a part of the Apache 2 Web server, but is now used in many other projects. Use of this library in Tomcat improves stability and performance, especially when Tomcat is used to connect to a native Web server like Apache.

If you see a message such as the following:

```
INFO: The Apache Tomcat Native library which allows optimal performance in
production environments was not found on the java.library.path
```

it is because the APR library is missing at Tomcat startup time. This is not an error, as the APR support is optional.

Locations for APR native library binaries for some platforms are listed at `http://tomcat.apache.org/tomcat-6.0-doc/apr.html`; for example, Windows binaries are available from `http://tomcat.heanet.ie/native/`. Download and save the `tcnative-1.dll` file from this Web site to the `<TOMCAT_HOME>/bin` directory. This DLL contains the native binaries for APR as well as for OpenSSL.

For Linux/Unix, you need to install this from source.

- If you don't have APR installed on your machine, download the source from `http://apr.apache.org`. Install APR using the following steps as the root user to install the APR in its default location (`/usr/local/apr`). If running as a non-root user, use the `--prefix` option with `configure` to specify another install location.

```
$ tar zxvf apr-x.y.z.tar.gz
$ cd apr-x.y.z
$ ./buildconf
$ ./configure
$ make install
```

The APR JNI wrapper sources are packaged, along with the Tomcat distribution, as the `tomcat-native.tar.gz` file in `<TOMCAT_HOME>/bin`. The installation steps are provided in the code that follows. In the steps shown, the default `apr_install_location` is `/usr/local/apr` as shown in the earlier step. If you don't need OpenSSL support, you can use the `--without-ssl` option with `configure` to disable it. If you need OpenSSL, you can download it from `www.openssl.org`. The JNI wrapper gets installed under `/usr/local/apr/lib`, and as before, a `--prefix` option for configure can install it elsewhere.

```
$ cd /path/to/tomcat/bin
$ tar zxvf tomcat-native.tar.gz
$ cd tomcat-native-x.y.z-src/jni/native
$ ./configure --with-apr=apr_install_location --with-ssl=openssl_install_location
$ make install
```

- After installing, add the APR directory to your `LD_LIBRARY_PATH` as shown:

```
$ export
   LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/apr/lib
```

When you restart Tomcat, you should now see an `INFO: Loaded Apache Tomcat Native library 1.1.9` message indicating that APR was detected and loaded. Chapter 10 explains the use of the APR Connector, which is an HTTP Connector for Tomcat developed using APR.

Troubleshooting and Tips

This final section deals with some common problems you may encounter after installing Tomcat. If you have further problems, you can find more information on the Tomcat Web site at the following URLs (as well as on various forums):

```
http://tomcat.apache.org/
http://java.sun.com/
```

You should also read the release notes available with each download.

Sometimes, when you attempt to launch Tomcat, the Tomcat window will appear briefly and then disappear. This usually occurs because of an error that causes Tomcat to crash and, thus, its window to disappear. The error message is also displayed, but because the window disappears so rapidly, the error cannot be seen.

If Tomcat does not start, it can be run in the current shell or as a command prompt (as opposed to a new pop-up window) so that the problem can be seen. To do this in Linux, type the following:

```
$CATALINA_HOME/bin/catalina.sh run
```

Or, in Windows, type the following:

```
%CATALINA_HOME%\bin\catalina run
```

This will produce the normal startup messages, and any errors will be displayed. These errors also appear in the `stdout.log` file in the `$CATALINA_HOME/logs` subdirectory.

Some common problems are discussed next.

Class Version Error

Tomcat 6 requires Java SE 5 or later to run. If an earlier version of Tomcat is used, you will see the following exception during Tomcat startup.

```
Exception in thread "main"
   java.lang.UnsupportedClassVersionError:
   org/apache/catalina/startup/Bootstrap (Unsupported
   major.minor version 49.0)
```

Check your `JAVA_HOME` settings, and make sure they point to a Java SE 5 installation.

The Port Number Is in Use

One possible error is that the chosen port is already in use. The error message will look similar to the following:

```
LifecycleException:  Protocol handler initialization failed:
java.net.BindException: Address already in use: JVM_Bind:8080
```

Tomcat uses port 8080 by default, as mentioned previously. You can determine whether another program is using this port by using the `netstat` utility on both Windows and Linux. Type `netstat -ao` on Windows and `netstat -lp` on Linux. Your shell/command prompt will list open ports on your system, which should indicate any process that is interfering with Tomcat. You have two options: shut the process down or change Tomcat's port as described earlier.

Running Multiple Instances

A common problem is trying to start a new Tomcat instance when one is already running. This is especially true if it's running as a daemon thread. Check to ensure that you aren't already running another instance of Tomcat.

A Proxy Is Blocking Access

If you have a proxy set up for all HTTP services, it may be blocking access to the server. You should bypass the proxy for all local addresses. Instructions are provided here.

In Firefox 2, Choose Tools➪Options. Next, choose Advanced, then Network Tab and then Settings. Select Manual proxy configuration and enter `localhost` and `127.0.0.1` in the No proxies for box.

In Internet Explorer, choose Tools➪Internet Options and then choose the Connections tab. Select the Lan Settings button and enter your proxy configuration by selecting the `Advanced` button in the window that opens. Enter `localhost` and `127.0.0.1` in the Exceptions box. This should work in most versions of Internet Explorer.

Summary

This chapter provided a great deal of information about selecting and installing a JDK and Tomcat in a variety of ways. Key points of this chapter include the following:

- In the majority of cases, installing the Tomcat server is a very straightforward process, because binary versions are available for the common platforms.
- The Tomcat installation directory structure includes seven important directories.
- Common installation problems include using the incorrect JVM version, the port number being in use, multiple instances running, and a proxy blocking access.

The next chapter examines Tomcat's architecture.

4

Tomcat Architecture

Tomcat is a powerful Web container that is made up of pluggable components that fit together in a nested manner. The configuration of these components dictates how the server will run, including such settings as whether specialized filters are used, which ports and address a server may listen on, whether it uses security or not, what the virtual hosts are, and much more. Tomcat can be easy to use in its standard configuration because you are using a default configuration. However, when you begin to think about using Tomcat in a production environment, you need to make changes to the configuration to fit your requirements. This is where having good knowledge of Tomcat's architecture comes in handy. Tomcat's configuration files can appear cryptic and difficult to follow or understand, but they follow how Tomcat pieces together its components. Understanding Tomcat's architecture can give you insight into the way Tomcat works and how its subcomponents make up a configuration.

This chapter explores Tomcat from an architectural perspective. The topics covered include:

- Tomcat directory structure
- Overview of the major Tomcat components
- Relationships between the components to make a full-service container

By the end of this chapter, you should have a comprehensive understanding of the Tomcat container architecture, its subcomponents, and their relationship to one another.

Tomcat Directory Overview

Before delving into the Tomcat architecture, understanding the Tomcat setup and directory layout will provide a solid understanding of where things live. When you download and uncompress the Tomcat bundle, it places all of its contents in a folder or directory named `apache-tomcat-6.X.XX` where *X.XX* represents the minor version numbers. This installation or top-level directory is what is known as the `CATALINA_HOME` and will be referenced as such from this point forward. When you download and install Tomcat, you are presented with the directory structure, as shown in Figure 4-1.

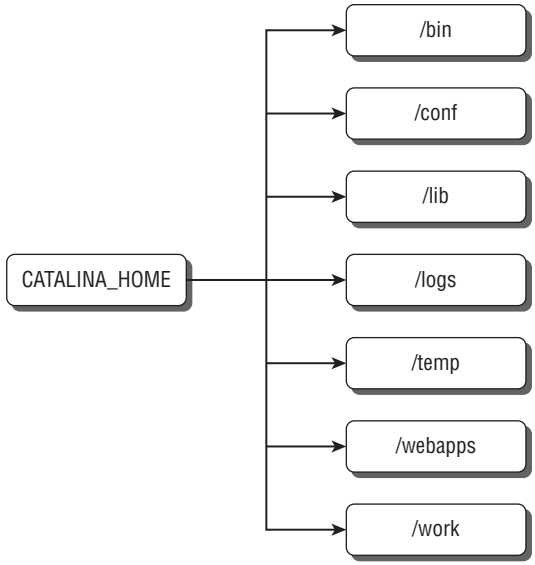

Figure 4-1: Tomcat directory structure

bin Directory

The `bin` directory contains the scripts and code required to execute the server. Depending on the version that you download, this directory may contain executable (`.exe`) files for a Windows service installation, or it will just contain the standard Java starting scripts. Both Unix/Linux shell scripts (`.sh`) and Windows batch (`.bat`) scripts exist here for the standard Tomcat download. Some JAR files also live in the bin directory, including `bootstrap.jar`, `commons-daemon.jar`, and `tomcat-juli.jar`. This directory is primarily used for starting and stopping Tomcat with either the startup or Catalina script. The `startup.sh(bat)` script is typically used for running Tomcat as a background process. Using the `catalina.sh(bat)` script allows you to have more fine-grained control over running Tomcat, such as running it as a foreground process, in debug mode. The `startup.sh(bat)` script actually runs `catalina.sh(bat)` and simply instructs it to run in the background.

conf Directory

The `conf` directory contains the files that are necessary to configure and set parameters for Tomcat when it executes. When Tomcat launches, it investigates the files in this directory and alters/creates the necessary objects and settings to run. The following is a list of the files along with a short description of their use.

- `catalina.policy`: Contains the security policy statements that are implemented by the Java SecurityManager. It replaces the `java.policy` file that came with your Java installation. It is used to prevent rogue code or JSPs from executing damaging code that can affect the container with calls such as `system.exit(0)`. It is used only when Tomcat is launched with the `-security` command-line parameter.
- `catalina.properties`: Contains lists of Java packages that cannot be overridden by executable Java code in servlets or JSPs, such as `java.*` or `org.apache.tomcat.*`, which could be a security risk. Also allows the setting to look for common JARs.

- `context.xml`: The common `context.xml` that is used by all Web applications. By default, this file is used to set up where to access the `web.xml` file in Web applications.
- `logging.properties`: The default logging configuration used by the JULI logger. By default, it uses a ConsoleHandler and FileHandler and sets up the logging level on a per-application or package basis.
- `server.xml`: The main configuration file for Tomcat. This file is used by the digester to "build" the container on startup to your specifications. This file follows the Tomcat architecture and will be discussed in more detail later in this chapter.
- `tomcat-users.xml`: Used for security to access the Tomcat administration applications. It is used with the default UserDatabase Realm as referenced in `server.xml`. All credentials are commented by default and should be changed if uncommented. The Tomcat Administration application isn't accessible until entries are uncommented or placed into this file.
- `web.xml`: The default `web.xml` file that is used by all Web applications. This `web.xml` sets up the JSPServlet to allow your applications to handle JSPs and a default servlet to handle static resources and HTML files. It also sets up the default session timeout and welcome files such as `index.jsp`, `index.htm`, and `index.html`; and it sets up default MIME types for the most common extensions.

In addition to these files, when an application is deployed to a Tomcat server, it creates the equivalent of a `context.xml` (usually named after the context path) with a structure following `[`*`engine name`*`]/[`*`host name`*`]/[`*`context path name`*`].xml`. As noted previously about the container-wide `context.xml` file, you can also create an `[`*`engine name`*`]/[`*`host name`*`]/context.xml.default` file where all Web applications under a specific host will adopt a set of default settings for the host's context.

lib Directory

The `lib` directory contains all of the JARs that are used by the container. This includes the Tomcat JARs and the Servlet and JSP application programming interfaces (APIs). This is the place where you place JARs that are shared across Web applications or JDBC JARs for connection pools.

logs Directory

This directory is used for the logging files that are produced while Tomcat is running. The JULI logging produces multiple files in this directory, and each log file is created for each day.

temp Directory

This is the temporary directory that Tomcat uses for scratch files and temporary use.

webapps Directory

The `webapps` directory is where your Web applications ultimately will live. If you are using an exploded WAR (a WAR file that is decompressed), it needs to be placed in this directory. Placing a WAR file in this directory also causes Tomcat to deploy the file. When you deploy a full WAR through the Manager console application or the Tomcat Client Deployer, your WAR file is also placed into this directory.

Tomcat comes standard with several applications that reside in this directory:

- `ROOT`: The welcome screen application. This is a special directory that designates the "`/`" root of your Web container. When you move Tomcat to a production environment you would likely remove this directory if your application reuses the "`/`" root context.
- `docs`: Contains the Tomcat documentation. It is the same documentation you would find on the `http://tomcat.apache.org/tomcat-6.0-doc` Web site.
- `examples`: Contains the JSP and servlet examples.
- `host-manager`: An application that allows you to manage the hosts that run in your application. Accessible from the `/host-manager/html` URL. It requires that you have set up proper credentials in the `conf/tomcat-users.xml` file to access this application.
- `manager`: An application that allows you to manage your applications in Tomcat. From this application you can start, stop, reload, deploy, and undeploy your applications. It is accessible from the `/manager/html` URL. It requires that you have set up proper credentials in the `conf/tomcat-users.xml` file to access this application.

work Directory

Directory for temporary and working files. This is heavily used during JSP compilation where the JSP is converted to a Java servlet and accessed through this directory.

An Overview of Tomcat Architecture

Tomcat's architecture was completely revised for version 4. It was rebuilt from the ground up because some users felt that the refactoring done in the previous Tomcat release, while improving its performance and flexibility, was always going to result in a somewhat limited server. A rather heated debate ensued regarding whether this was actually the case. The result of this controversy was the 3.2 architecture branching from the main development tree in a continued refactoring effort, leaving the 4.0 version to become the main focus of the project.

Tomcat 6 is the latest iteration of the Tomcat 4 architecture. Tomcat 6 supports the latest Servlet and JSP specifications, versions 2.5 and 2.1, respectively.

Tomcat 6 consists of a nested hierarchy of components. Some of these components are called *top-level components* because they exist at the top of the component hierarchy in a rigid relationship with one another. *Containers* are components that can contain a collection of other components. Components that can reside in containers, but cannot themselves contain other components, are called *nested components*. Figure 4-2 illustrates the structure of a typical Tomcat 6 configuration.

This diagram represents the most complete topology of a server. However, you should be aware that some of these objects can be removed without affecting the server's performance. Notably, the Engine and Host may be unnecessary if an external Web server (such as Apache) is carrying out the tasks of resolving requests to Web applications. This topology also allows components to be created and removed on-the-fly, or while the container is running. This is evidenced through the host-manager application, where you can add and remove Host objects at will. This is further demonstrated through the manager application where you are able to stop, start, deploy, and undeploy Web applications, which literally are Context objects.

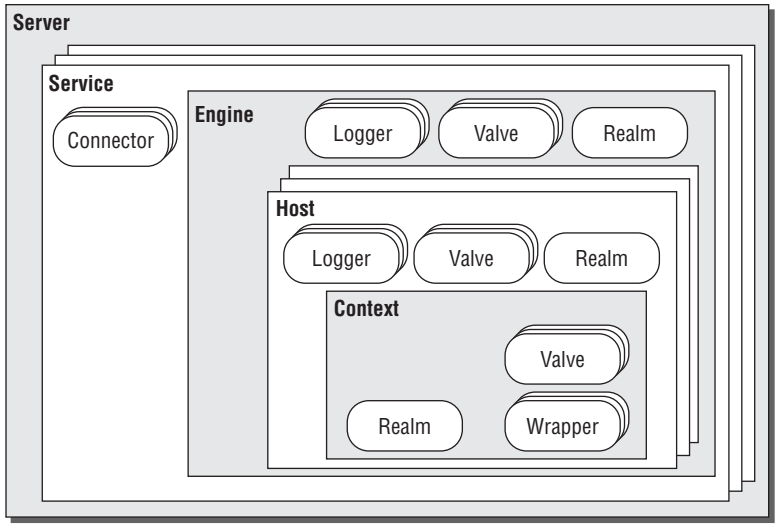

Figure 4-2: Tomcat's architecture

Here, components that can be contained multiple times are denoted by a symbol that has multiple profiles, including Connector, Logger, Valve, Host, and Context.

The nested relationships of the components in this architecture are parent-child in nature. This essentially means that each component may have one or more child components, and those children may have children of their own. For example, as shown in Figure 4-2, you can have one or more Service objects as children. Each Service object may contain a single Engine and one or more Connector objects. Each Engine may have one or more Host objects as children, and so on. The Wrapper objects in the Context represent the holders for servlets and JSPs.

The following sections examine each component at a high level. Chapter 5 discusses each component's configuration.

The Server

The *Server* is Tomcat itself — an instance of the Web application server — and is a top-level component. It owns a port that is used to shut down the server. In addition, the Server can be set in debug mode, which instantiates a version of the Java Virtual Machine (JVM) that enables debugging.

Only one instance of the Server can be created inside a given Java Virtual Machine (JVM).

Separate Servers configured to different ports can be set up on a single machine to separate applications so that they can be restarted independently. That is, if one Server running in a JVM were to crash, the other applications would be safe in another Server instance. This is sometimes done in hosting environments in which each customer has a separate instance of a JVM, so a badly configured/written application will not cause others to crash.

The server is an implementation of the `Server` interface. The default and usual configuration is for Tomcat to implement the `StandardServer` object for the server.

The Service

A *Service* groups a container (usually of type Engine) with a set of Connectors and is a top-level component.

> *An Engine is a request-processing component that represents the Catalina Servlet engine. It examines the HTTP headers to determine the virtual host or context to which requests should be passed.*

Each Service represents a grouping of Connectors (components that manage the connection between the client and server) and a single container, which accepts requests from the Connectors and processes the requests to present them to the appropriate Host. Each Service is named so that administrators can easily identify log messages sent from each Service.

In other words, the container contains the Web applications. It is responsible for accepting requests, routing them to the specified Web application and specific resources, and returning the result of the processing of the request. Connectors stand between the client making the request and the container. They provide additional services such as SSL support.

Multiple Service objects may be children to the Server. However, most of the time, you will use only a single Service in a container. Tomcat typically uses the `StandardService` object as the service that implements the `Service` interface.

The Connectors

Connectors connect the applications to clients. They represent the point at which requests are received from clients and are assigned a port on the server. The default port for non-secure HTTP applications is kept as 8080 to avoid interference with any Web server running on the standard HTTP port, but there is no reason why this cannot be changed as long as the port is free. Multiple Connectors may be set up for a single Engine or Engine-level component, but they must have unique port numbers.

> *The default port to which browsers make requests if a port number is not specified is port 80. If Tomcat is run in standalone mode, the port for the primary Connector of the Web application can be changed to 80 by reconfiguring this component.*

The default Connector is Coyote, which implements HTTP 1.1; Tomcat also comes with an AJP connector. In addition, the HTTP connector can be used with SSL as well. Both the HTTP and AJP connectors are fully supported by Tomcat. However, there are alternative Connectors, such as the old JServ and JK2, which work, but are no longer supported. These are discussed later in this chapter.

The Engine

The next component in the architecture is the top-level container — a container object that cannot be contained by another container. This means that it is guaranteed not to have a parent container. It is at this level that the objects begin to aggregate child components.

> *Strictly speaking, the container does not need to be an Engine; it just has to implement the container interface. This interface mandates the following: that the object implementing it is aware of its position in the hierarchy (it knows its parent and its children), that it provides access to logging, that it provides a Realm for user authentication and role-based authorization, and that it has access to a number of resources, including its session manager (and some internally important aspects that you do not need to worry about).*

In practice, the container at this level is usually an *Engine* and so it makes sense to discuss it in that role. As mentioned previously, an Engine is a request-processing component that represents the Catalina Servlet engine. It examines the HTTP headers to determine the virtual host or context to which requests should be passed.

When the standalone configuration is used, the Engine that is used is the default one. This Engine does the checking mentioned earlier. When Tomcat is configured to provide Java servlet support for a Web server, the default class used to serve requests is overridden because the Web server has typically determined the correct destination of the request.

The host name of the server to which the Engine belongs is set as a property in multi-homed machines. An Engine may contain Hosts representing a group of Web applications and Contexts representing a single Web application.

The Realm

The Realm for an Engine manages user authentication and authorization. During the configuration of an application, the administrator sets the roles that are allowed for each resource or group of resources, and the Realm is used to enforce this policy.

> *Realms can authenticate against text files, database tables, LDAP servers, and the Windows network identity of the user. You learn more about this in Chapter 14.*

A Realm applies across the entire Engine or top-level container, so applications within a container share user resources for authentication. This means that, for example, a manager for the intranet will have the same rights as the manager of the e-commerce site should both these applications be in the same Engine.

The Realm may also be applied as a child at the Host, or even more specifically a Context. This means that the Realm, when applied at a Host level, is used by all Contexts (Web applications) that are associated with the Host.

The Realm is an object that may be superceded by its children objects. This means a Realm may be attached to an Engine so that all of its children will inherit it. But if one of its children declares a Realm of its own at a Host or Context level, then that object will use its own Realm, instead of the parent.

By default, a user must still authenticate separately to each Web application on the server. You will see how this can be changed in Chapter 6, using single sign-on, but, in brief, this is implemented as a Valve in Tomcat.

The Valves

Valves are components that enable Tomcat to intercept a request and preprocess it. They are similar to the filter mechanism of the Servlet specifications, but are specific to Tomcat. Hosts, Contexts, and Engines may contain Valves. A Valve is essentially a super-filter, very similar to a servlet filter, but it intercepts and invokes at a much higher level.

Valves are commonly used to enable single sign-on for all Hosts on a Server, as well as log request patterns, client IP addresses, and server usage patterns (peak traffic, bandwidth use, mean average requests per time unit, the resources that most requests ask for, and so on). This is known as *request dumping*, and a *request dumper valve* records the header information (the request URI, accepted languages, source IP,

host name requested, and so on) and any cookies sent with the request. Response dumping logs the response headers and cookies (if set) to a file.

Valves are typically reusable components, and can therefore be added and removed from the request path according to need. Their inclusion is transparent to Web applications, although the response time will increase if a valve is added. An application that wishes to intercept requests for pre-processing and responses for post-processing should use the *filters* that are a part of the Servlet specifications.

A Valve may intercept a request between an Engine and a Host/Context, between a Host and a Context, and between a Context and a resource within the Web application.

You may have one or more Valves at its particular parent, and the Valves are typically chained in the order that they were added to the parent. This means that you may have Valves that depend on previous Valves as long as they are added in a specific order. The `server.xml` file will install the Valves in the order that they are found in this file.

The Loggers

Loggers report on the internal state of a component. They can be set for components from top-level containers downward, except for the Context. Logging behavior is inherited, so a Logger set at the Engine level is assigned to every child object unless overridden by the child. The configuration of Loggers at this level can be a convenient way to decide the default logging behavior for the server.

This establishes a convenient destination for all logging events for those components that are not specially configured to generate their own logs.

The Host

A *Host* mimics the popular Apache virtual host functionality. In Apache, this enables multiple servers to be used on the same machine, and to be differentiated by their IP address or by their host name. In Tomcat, the virtual hosts are differentiated by a fully qualified host name. Thus, the Web sites `www.example.com` and `www.example.net` can both reside in the same server, with requests for each routed to different groups of Web applications.

Configuring a Host includes setting the name of the host. The majority of clients can be depended on to send both the IP address of the server and the host name they used to resolve the IP address. The host name is provided as an HTTP header that an Engine inspects to determine the Host to which a request should be passed.

The Host is a child of the Engine. The Engine may contain one or more host objects that will represent a default host and zero or more virtual host objects. The Engine has a setting to designate one of the host objects as the default. This is so it can dispatch requests to a host if it is unable to resolve who it is intended for.

The Context

Finally, there is the Web application, also known as a *Context*. Configuration of a Web application includes informing the Engine/Hosts of the location of the root folder of the application. Dynamic reloading can also be enabled so that any classes that have been changed are reloaded into memory. However, this is resource-intensive, and is not recommended for deployment scenarios.

The Context may also include specific error pages, which enable a system administrator to configure error messages that are consistent with the look and feel of the application, and usability features (such as a search engine, useful links, or a report-creating component that notifies the administrator of errors in the application).

A Context can also be configured with initialization parameters for the application it represents and for access control (authentication and authorization restrictions). Chapter 14 provides more information on these two aspects of Web application deployment.

A Context implements the `Context` interface. Most Context implementations are created with the `StandardContext` class.

Because the Context is itself a container at the Web application level, it then becomes a parent of servlets and filters. Obviously, you may have zero, one, or more servlets in an application, and the contexts add these children as `StandardWrapper` objects.

The Remaining Classes in the Tomcat Architecture

Tomcat also defines classes for representing a request, a response, a session that represents a virtual connection between a server and a client, and listeners. Tomcat also has components that allow you to swap out the session manager, which carries with it a slew of clustering components as well. Some of these classes are described in detail in the remainder of the book.

> *Listeners listen for significant events in the component they are configured in. Examples of significant events include the instantiation of the component and its subsequent destruction.*

Connector Architecture

All the Connectors work on the same principle. They have an Apache module end (`mod_jk`, `mod_jk2`, or `mod_proxy`) written in C that Apache (or other supported Web servers) loads just like the other Apache modules.

On the Tomcat end, each Web container instance has a Connector module component written in Java. In Tomcat 6.*x*, this is with the `org.apache.catalina.Connector` class. The class constructor takes one of two connector types, HTTP/1.1 for direct HTTP/HTTPS calls, or AJP/1.3 for AJP. If you have a specific custom-made protocol, you can pass the entire package and class name to the Connector and it will use that instead. However, for nearly all cases, the HTTP and AJP protocols fit the need. Now you may be wondering why you do not call the Connector constructor. You do indirectly in your `server.xml` file in the `Connector` tag with the `protocol` attribute. This value is what gets passed into the Connector's constructor when Tomcat is initialized. When you pass the HTTP or AJP strings, they are mapped to the following classes.

If Apache Portable Runtime (APR) is supported:

- **HTTP/1.1:** `org.apache.coyote.http11.Http11AprProtocol`
- **AJP/1.3:** `org.apache.coyote.ajp.AjpAprProtocol`

If APR is not supported:

- **HTTP/1.1:** org.apache.coyote.http11.Http11Protocol
- **AJP/1.3:** org.apache.jk.server.JkCoyoteHandler

The Apache Portable Runtime is a set of native libraries that attach a communication layer to the operating system that provides a common and standard API that leverages the best components of the Apache Web server to give high scalability to other projects that use it. It basically enables a level of scalability from Tomcat's container that nears what you would get from a native Web server. Enabling APR is a matter of building and installing the proper APR and native libraries, and Tomcat will use these automatically. Installing the APR libraries is beyond the scope of this section. You can find more information on this topic at `http://tomcat.apache.org/tomcat-6.0-doc/apr.html`. This section concentrates only on the pure Java implementation of the HTTP and AJP connectors.

Communication Paths

The Web server handles all requests for static content, as well as all non-servlet/JSP dynamic content (CGI scripts, for example). When it comes across content targeted for the Servlet container, the Web server passes the request to the module in question (that is, `mod_jk`, `mod_proxy`, and so on). The Web server knows what content to pass to the Connector module because the directives in the Web server's configuration specify this.

To illustrate, if the JK Connector is being used, Apache's configuration would have entries similar to the following:

```
# Configuration directives in Apache's httpd.conf for mod_jk
# Send all request for JSP (extension *.jsp) or
# servlets (web application path /servlet) to the
# AJP Connector
LoadModule    jk_module  {path-to-modules}/mod_jk.so
AddModule     mod_jk.c
JkMount /*.jsp ajp13
JkMount /myapp/* ajp13
```

Alternatively, if `mod_proxy` is being used, the entries would be similar to the following:

```
# Configuration directives in Apache's httpd.conf for mod_proxy
LoadModule proxy_module {path-to-modules}/mod_proxy.so
LoadModule proxy_balancer_module {path-to-modules}/mod_proxy_balancer.so
LoadModule proxy_ftp_module {path-to-modules}/mod_proxy_ftp.so
LoadModule proxy_http_module {path-to-modules}/mod_proxy_http.so
LoadModule proxy_connect_module {path-to-modules}/mod_proxy_connect.so
AddModule    mod_proxy.c
ProxyRequests off
ProxyPassReverse /myapp http://localhost:8080/myapp
ProxyPass /myapp http://localhost:8080/myapp
```

The Connector module then sends the request encoded in a manner specific to a protocol (AJP or HTTP) over a network connection to a Connector. (There can be more than one instance of the Servlet container in the backend.) The Connector gets the request serviced by the Servlet container, and sends the response back to the Apache module. Figure 4-3 illustrates this.

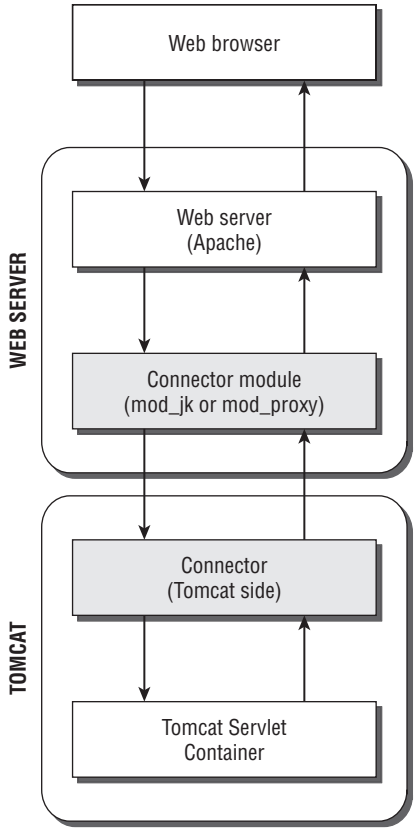

Figure 4-3: Communication between Connector and Apache modules

Although you can use both `mod_jk` and `mod_proxy`, a recent movement calls for the use of `mod_proxy` as a standard. `mod_jk` has been around for a while and at one point `mod_jk2` was a thriving project, but over time its support has slipped and now it is considered unsupported. However, `mod_jk` is still supported in Tomcat. `mod_jk` performs slightly better than `mod_proxy` does, but a new module is available that offers the best of both worlds: `mod_proxy_ajp`. You can find more information on this module at `http://httpd.apache.org/docs/2.2/mod/mod_proxy_ajp.html`.

Connector Protocols

Tomcat has two protocols for its Web server Connectors: AJP and HTTP. These protocols essentially define the (binary) format for the data transmitted between the Web server and Tomcat and the control commands.

AJP Protocol

The *Apache JServ Protocol (AJP)* is a historical name. AJP10 and AJP11 were the protocols that the now-obsolete JServ Connector implemented. The current version of AJP is AJP13, and is implemented by the JK and JK2 Connectors.

AJP13 uses a binary format for transmitting data between the Web server and Tomcat. The earlier versions of AJP (AJP10 and AJP11) used a text-based data format. The communication between the Web server and the Servlet container is over a network socket, and the same connection is reused for multiple requests and responses. The connection is made persistent for better performance. However, once a connection has been assigned to a particular request, it is not assigned to any other request until the request-handling cycle is complete.

The AJP packet format consists of a packet header and the actual payload. The packet header indicates the payload size. The type of message is in the first byte of the payload. The message could be an HTTP request/response packet, or even a control command (for example, the Web server asks the Servlet container to shut down). The protocol defines binary encoding for the HTTP commands and headers. (For example, the `GET` command is represented by the byte value `2`.)

Figure 4-4 shows the AJP packet structure from the Web server side to the Servlet container. Figure 4-5 shows the packet structure from the Servlet container to the Web server.

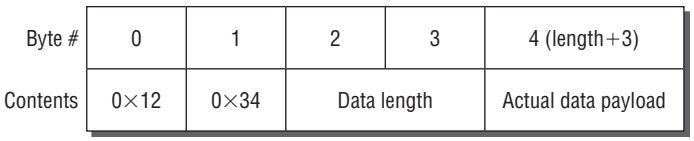

Figure 4-4: AJP packet structure from the Web server side to the Servlet container

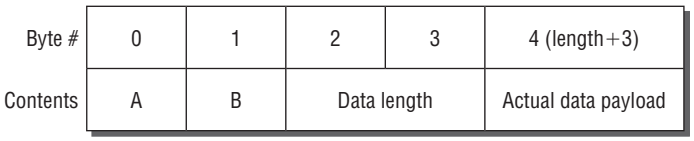

Figure 4-5: AJP packet structure from the Servlet container to the Web server

As you can see, the binary data packet from the Web server side starts with the sequence 0x1234. This is followed by the packet size (2 bytes) and then the actual payload data. On the return path, the packets are prefixed by AB (the ASCII codes for A and B). The protocol then defines the structure for the payload data (the type of message is in the first byte of the payload, and so on).

Administrators or Web developers do not need to know the AJP protocol and its packet structure. The specifics of the AJP protocol and packet structure are relevant only to people who are working on (or curious about) Tomcat internals, or who are interested in implementing a Connector for AJP. Users of Tomcat should have no reason to implement a Connector.

For further information on the details of AJP, see the documentation that comes with the JK Connector source code or visit the Web site at `http://tomcat.apache.org/tomcat-6.0-doc/config/ajp.html`.

HTTP Protocol

The HTTP protocol is exactly as the name implies; it is the conduit to communicate using the HTTP protocol. This covers both HTTP and secure HTTPS. It basically allows Tomcat to work as a fully functional

Web server on its own. The Connector is typically listening on a particular port for HTTP traffic and may be configured also to respond on only particular IP addresses. This connector can also be used to process HTTPS protocol by setting the Connector's `secure` attribute to `true` in the `server.xml` file. However, setting this to `true` forces you to set up an SSL certificate and have the Connector filled with several other attributes. Setting this up is covered in Chapter 10.

Choosing a Connector

Choosing a Connector on the Web server side was very confusing in earlier versions of Tomcat. In Tomcat 6.*x*, sticking to JK or Proxy simplifies the task. The use of the other major Connector (`webapp`) is deprecated. For historical interest (or for those examining such deployments), the following sections introduce all Tomcat Connectors.

AJP History

Apache JServ (`http://java.apache.org/jserv/`) was a Servlet engine that implemented the JavaSoft Servlet API, version 2.0. (The current version of the API is 2.4.) JServ is now in maintenance mode, and has been superseded by Tomcat. JServ did not have an HTTP stack, and so it came with a Connector (`mod_jserv`) that used Apache (versions 1.2 and 1.3) for this. The `mod_jserv` module also works as a Connector for Tomcat 3.*x* and earlier versions.

`mod_jserv` defined the AJP protocol that specified the packet format for communication between the Apache and Tomcat ends of the Connector. `mod_jserv` implemented versions AJP11 and AJP12 of this protocol. This protocol lives on in the JK and JK2 Connectors.

AJP JK

The JK Connector is a cleaned-up version of the JServ Connector, and has a refactored code base. It implements the same protocol (AJP) that JServ did. The versions that it supports include AJP11, AJP12, and AJP13.

JK adds support for many more Web servers than JServ supported, including Apache 1.3 and 2.*x*, Netscape, Domino, AOLServer, and IIS. On the Servlet side, it supports Tomcat 3.*x*–6.*x*, and JServ. It also supports redirection of incoming requests, and thus can be used to achieve load-balanced request sessions.

The JK Connector renders `mod_jserv` obsolete. It offers a less complex configuration and better support for SSL. (For example, `mod_jserv` couldn't reliably differentiate between HTTP and HTTPS requests.) If you are looking for a stable Connector that supports a large(r) number of Web servers and Tomcat versions, `mod_jk` is the Connector for you.

webapp

The `webapp` Connector implements the WARP protocol for connecting Tomcat and Apache. This protocol has built-in support for auto-deployment and Web-application configuration.

`webapp` uses the Apache Portable Runtime (APR) library for operating-system portability, so it can only be used with Apache 1.3 and 2.0. It is limited also by the versions of Tomcat it supports (currently only Tomcat 4.*x*).

Furthermore, it is limited in the features it supports. The current version of `webapp` does not support load balancing and fault-tolerance, and has known problems in its Windows support.

`webapp` implements the WARP protocol, which was designed with performance as a major consideration. However, it was an experimental Connector, and has a number of known bugs. Its use has been deprecated, so this Connector is not discussed in this book.

JK2

JK2 was the next generation for the JK Connector and implements the AJP13 protocol. It supports all the Web servers that `mod_jk` does (Apache 1.3/2.0, Domino, Netscape, AOLServer, and IIS). It also supports the Tomcat 3.*x*–6.*x* Servlet containers.

The JK2 Connector improves on the JK Connector in many ways, including the following:

- Developed with Apache 2.0 in mind, it works with Apache 1.3 as well.
- It is better suited than JK for multithreaded Web servers such as IIS and iPlanet.
- It is more modular than JK.
- It supports fast UNIX sockets.
- It can be extended to support other communication channels.
- It is better suited for JNI.
- It offers support for monitoring.

Tomcat 4.1 and 5.*x* shipped with a JK2 Connector called Coyote jk2. Coyote is a new architecture/API for the Java code that talks to the Connectors. Tomcat 4.1 and 5.*x* have a Coyote HTTP/1.1 Connector (discussed in Chapter 11) in addition to the Coyote jk2 Connector.

JK2 supersedes the JK Connector in Tomcat versions 4.*x* and 5.*x*; however, it is no longer supported so you should use the JK connector instead. Therefore, for Tomcat 6, consider using JK.

Proxy

`mod_proxy` is used to proxy HTTP requests from the Web server to Tomcat and back. The `mod_proxy` module is able to take an HTTP request, examine its URL, and decide whether the inbound request is for Tomcat or not by matching based on the URL. If it's found, it passes the request on to Tomcat and holds the sockets open for both the client and Tomcat. Tomcat receives the request, just as if it were received on the port as normal. Tomcat returns the response to the Web server, and the request is passed on to the client and the sockets are closed. Essentially, the Apache Web server acts as a middleman between the client and Tomcat. The `mod_proxy` is also sure to translate the URLs between, for example, `localhost:8080` and `www.example.com`, assuming Tomcat is running on `localhost`, and the client resolves the Web server as `www.example.com`.

Lifecycle

You have seen the components in Tomcat and how they relate to one another. With this pluggable-style architecture, there needs to be a way for each of these components to know when to start and stop. When Tomcat starts, the Service object starts, and its subcomponents or children also need to start. The same is true when the container stops. Hence, each parent must ensure that each child is started and stopped. As you can see, there is somewhat of a chain reaction, or domino effect, to a top-level parent component

starting/stopping, and all of the children receiving this event. This is done through the `Lifecycle` interface: `LifecycleEvent` and `LifecycleListener`.

Lifecycle Interface

A majority of the Tomcat components implement the `Lifecycle` interface. The key methods in this interface enforce the implementation of `start()` and `stop()` and also allow the implementation of classes to add listeners that can be interested in specific events. The following listing shows the `Lifecycle` interface with the methods and events.

```
public interface Lifecycle {
    public static final String INIT_EVENT = "init";
    public static final String START_EVENT = "start";
    public static final String BEFORE_START_EVENT = "before_start";
    public static final String AFTER_START_EVENT = "after_start";
    public static final String STOP_EVENT = "stop";
    public static final String BEFORE_STOP_EVENT = "before_stop";
    public static final String AFTER_STOP_EVENT = "after_stop";
    public static final String DESTROY_EVENT = "destroy";
    public static final String PERIODIC_EVENT = "periodic";
    public void addLifecycleListener(LifecycleListener listener);
    public LifecycleListener[] findLifecycleListeners();
    public void removeLifecycleListener(LifecycleListener listener);
    public void start() throws LifecycleException;
    public void stop() throws LifecycleException;
}
```

Nearly all of the Tomcat components implement the `Lifecycle` interface. The major components usually contain a `LifecycleSupport` object that manages all of the `LifecycleListener` objects for that component. The `LifecycleSupport` is what propagates and fires the general events in the code shown in the following listing, as well as other events. When the top-level component is started, it calls all of its children's start methods. Those children then call their children's start method, and so on. This effectively starts all components automatically when they implement the `Lifecycle` interface. The reverse is true when stopping a component. The beauty of this architecture is that when you stop a parent, all of its children are stopped as well. This is why in the host-manager application, you are able to start and stop hosts at will, without any impact on other running hosts.

LifecycleListener Interface

The `LifecycleListener` is an interesting interface because you can add listeners at any level in the Tomcat container (Server, Service, Engine, Host, or Context) that can execute specific code when a particular event is fired. The default Tomcat configuration contains three listeners that are declared at the server level. You can write your own listeners to execute your own code when an event occurs, and you then declare these listeners in `server.xml` or `context.xml` at the specific level. Your listener implements the `LifecycleListener` as shown here:

```
public interface LifecycleListener {
    public void lifecycleEvent(LifecycleEvent event);
}
```

The `lifecycleEvent()` method takes a `LifecycleEvent` object, which is a wrapper for the event to contain the event type as well as additional payload data that may be associated with the event. For example, let's say you want to receive an e-mail every time the Host starts or shuts down. You can write a listener similar to the following:

```
public MyListener implements LifecycleListener {
    public void lifecycleEvent(LifecycleEvent event){

        if (Lifecycle.START_EVENT.equals(event.getType())){
           //Code here to email that a Start Event was received
        }
        if (Lifecycle.STOP_EVENT.equals(event.getType())){
          //Code here to email that a Stop Event was received
        }
    }
}
```

The payload data could be used if you were to write your own events that are a part of your own components (engine, host, or context) and fire specialized events of your own.

As you can see, the plug-ability of the Tomcat container is extremely powerful as any components that work with the lifecycle interfaces can become an integrated part of the Tomcat infrastructure immediately.

Configuration by Architecture

Throughout this chapter, you have seen the different components of the Tomcat architecture and the parent-child relationships that they build upon each other. If you understand these relationships, configuring Tomcat is much easier.

The most important and critical configuration file in Tomcat is `<TOMCAT_HOME>/conf/service.xml`. When Tomcat starts, it uses a version of the Apache Commons Digester to read the `service.xml` file. The Digester is a utility that reads XML files and creates Java objects from a set of rules. Thus, it reads `service.xml` and creates the Tomcat components and sets properties based upon what is in this file, as shown in the sample listing:

```
<Server port="8005" shutdown="SHUTDOWN">
    <Listener className="org.apache.catalina.core.AprLifecycleListener"
        SSLEngine="on" />
    <Listener className="org.apache.catalina.core.JasperListener" />
    <Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" />
    <Listener className="org.apache.catalina.mbeans.
        GlobalResourcesLifecycleListener" />
    <Service name="Catalina">
        <Connector port="8080" protocol="HTTP/1.1"
                maxThreads="150" connectionTimeout="20000"
                redirectPort="8443" />
   <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
   <Engine name="Catalina" defaultHost="localhost">
        <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                resourceName="UserDatabase"/>
```

```
        <Host  name="localhost" appBase="webapps"
               unpackWARs="true" autoDeploy="true"
              xmlValidation="false" xmlNamespaceAware="false"/>
      </Engine>
     </Service>
  </Server>
```

The very nature of XML lends itself to parent-child and nested relationships. The Tomcat development team decided to make this a direct correlation between the `service.xml` and the Tomcat architecture. In the previous listing, a stripped-down version of the `service.xml` is shown. Notice how the XML follows the Tomcat architecture exactly. The server contains a service, which contains two Connectors and an Engine. The Engine contains a Realm and a Host. Also, note that each of the attributes in the XML represents properties in the underlying object. Therefore, the `service.xml` is truly a representation of the Tomcat architecture.

This concept is also followed in the `context.xml`, where you can set different session managers, and so on, and this also follows the architecture. This should help you understand and configure very complex Tomcat implementations with ease by remembering how the Tomcat architecture works.

Summary

To conclude this chapter on Tomcat's architecture, let's review some of the key points discussed:

- The Tomcat architecture represents components that are nested. Most of the components implement parent-child and nested relationships.
- The Tomcat components implement the `Lifecycle` interface, which allows subcomponents to be started and stopped without affecting other areas of Tomcat.
- The `service.xml` file's parent-child relationships follow the Tomcat architecture of its components. Therefore, having a strong comprehension of the Tomcat architecture will simplify the configuration of Tomcat.

Now you should be comfortable with Tomcat's architecture. In Chapter 5, you examine the steps for configuring Tomcat.

5

Basic Tomcat Configuration

The focus of this chapter is on the basic configuration of Tomcat 6. The Tomcat 6 server is configured by setting parameter values for the components of Tomcat 6 before starting the server. All architectural components (such as Service, Servers, Engine, Connectors, Realm, and Valve) can be configured. This chapter describes how to configure these components, the range of allowed parameter values, and how they affect Tomcat's operation.

Tomcat 6 configuration information is always stored within one or more XML files. Numerous XML elements are in these files, and each element has attributes that correspond to a configurable aspect of a Tomcat 6 architectural component (see Chapter 4 for more information about Tomcat 6 architecture). This chapter examines each of these configurable attributes, and describes how they affect the behavior of the corresponding Tomcat 6 components.

This chapter provides a detailed, line-by-line explanation of the most important Tomcat 6 configuration files, including the `server.xml` file (the primary configuration file for Tomcat server instance) and `context.xml` file (the server configuration file for each Web application).

Special attention is paid to the default Tomcat 6 configuration in this chapter's coverage. This default configuration exists in the form of a set of default configuration files that are included with the Tomcat 6 distribution. If you start Tomcat 6 without first editing any of the XML configuration files, this is the configuration that is used. Incidentally, it is also the *bootstrap* configuration that is used if you start up Tomcat 6, after enabling the administration tool, to make configuration changes. Therefore, it is important to understand what this special bootstrap configuration will do, and how you may be able to modify it for specific production environments.

This chapter also touches on some advanced configuration topics (such as Realm configuration and the configuration of fine-grained security policy control over Tomcat 6 server instances), but detailed descriptions of these concepts are provided in later chapters.

Tomcat 5.5 and earlier versions can also be configured using a Web-based GUI configurator called the admin Web application. Although this application has not yet been made available for Tomcat 6, this chapter includes an overview of its capabilities.

By the end of this chapter, you will be fluent with basic Tomcat 6 configuration, and will be able to configure Tomcat 6 by editing the XML configuration files.

You will also be completely familiar with the basic (default) configuration of the Tomcat 6 server, and will be able to modify this configuration for your own production needs.

Tomcat 6 Configuration Essentials

A Tomcat 6 server instance reads a set of configuration XML files upon startup. To configure a Tomcat 6 server instance, it is necessary to modify these XML files. The following table shows the files that affect the behavior of the Tomcat 6 instance.

File Name	Description
`server.xml`	Primary configuration file for the Tomcat server components. This includes the configuration of Service, Connector, Engine, Realm, Valves, Hosts, and so on.
`context.xml`	Default version of the per-application configuration file for server components. Any components configured in this default file apply to all applications running on the server. An individual application can override the global configuration by defining its own `context.xml` file (placed in the `META-INF` directory of the application).
`web.xml`	Default version of the standard Java EE deployment descriptor for Web applications. This is used by Tomcat 6 for all automatically deployed Web applications, or applications without their own specific deployment descriptor. If a Web application has its own deployment descriptor, its content will always override the configuration settings specified in this default descriptor.

Tomcat 6 looks for these configuration files in a specified configuration directory. This configuration directory is specified via an environment variable. Tomcat 6 first checks the `$CATALINA_BASE` (`%CATALINA_BASE%` on Windows) environment variable. If this environment variable is defined, Tomcat 6 looks in the specified directory for a `conf` subdirectory. The configuration files are expected to reside in this `conf` subdirectory. It is quite straightforward to configure multiple, concurrently running Tomcat 6 instances on the same physical machine. This can be done by specifying different `$CATALINA_BASE` directories for each Tomcat instance. Typically, this is performed using a different shell script or batch file to set the `$CATALINA_BASE` variable and start each instance.

If `$CATALINA_BASE` is not specified, the `$CATALINA_HOME` (`%CATALINA_HOME%` on Windows) environment variable is used instead. The `$CATALINA_HOME` environment variable is a required variable that specifies where Tomcat 6 is installed on your system. In fact, the `$CATALINA_HOME` variable is used to locate the executables (i.e., Catalina, Jasper, and Coyote) to run Tomcat. In this case, Tomcat 6 will look into the `conf` directory under `$CATALINA_HOME` for the server configuration files. This behavior is compatible with older versions of Tomcat servers. However, if you need to run multiple instances on the same machine (using `$CATALINA_HOME`), you must duplicate the entire Tomcat 6 distribution (including the large `bin` and `lib` directories, and so on).

In the rest of this chapter, references to the `$CATALINA_HOME` variable can be taken to mean `$CATALINA_BASE` if you are using Tomcat 6's multi-instance support.

On a per-application level, `META-INF/context.xml` *can be used to configure additional server components, and* `WEB-INF/web.xml` *can custom-configure the application for deployment.*

Files in $CATALINA_HOME/conf

In the `$CATALINA_HOME/conf` directory of the Tomcat 6 server distribution, you will see several configuration files. Following is a brief synopsis of each of these files:

- `server.xml`: This is the main configuration file for the Tomcat server and is the one that Tomcat actually reads at startup. By default, it contains a configuration that is ready to run on your machine immediately. Components configured in this file affect the entire Tomcat instance. Virtual hosts are also configured in this file. Application-level context configuration should not be made in this file; use a per-application `context.xml` instead. Before you modify `server.xml`, the best practice is to first back up the existing file and add a suffix to the backups with the current date and time before you perform the backup.
- `tomcat-users.xml`: This file contains user authentication and role-mapping information for setting up a UserDatabase Realm. Tomcat's `manager` applications use this file by default. UserDatabase Realm is a component in Tomcat 6 used to implement a database of users/passwords/roles for authentication and container-managed security. To add/remove users, or assign/unassign roles to existing users, edit this file.
- `context.xml`: This is a default application context configuration for any Web applications that are deployed on this Tomcat server instance. Components such as session persistence managers, Realms, and resources such as JDBC connections can be configured in `context.xml`. If the application does not define its own `context.xml`, the configuration in this default instance applies. Applications can customize or override component configurations via their own `context.xml`.
- `web.xml`: This is a default deployment descriptor file for any Web applications that are deployed on this Tomcat server instance. It provides basic servlet definition and MIME mappings common to all Web applications, and also acts as the deployment descriptor for any Web application that does not have its own deployment descriptor.
- `catalina.policy`: Java SE has a fine-grained security model that enables the administrator to control in detail the accessibility of system resources. This is the default policy file for running Tomcat 6 in secured mode. This is covered in detail later in this chapter.
- `catalina.properties`: Tomcat 6 reads and uses the properties value in this file upon startup. It provides for internal package access and definition control, as well as control over contents of Tomcat class loaders (see Chapter 9).
- `logging.properties`: Tomcat 6 uses its own implementation of Java Logging to write logs during operations. This is the configuration file for logging. You can control the level of log produced on a per-component basis, and can configure the destination of logging (to a file, to the console screen, and so on).

Basic Server Configuration

This section provides a line-by-line analysis of the default `server.xml` file. This file is created as an XML 1.0 document; it is assumed that you are familiar with XML.

Server Configuration via the Default server.xml

The default server configuration is stored in the `server.xml` file included with the distribution. If you start and run Tomcat 6 without your own customization, the configuration in this `server.xml` is used. As such, it is coded to be as simple as possible, sufficient to run the set of example Web applications that comes with Tomcat distribution. Tomcat 6 should work with most existing Tomcat 5.5.*x* `server.xml` configurations, although you should re-inspect these configurations and determine whether additional Tomcat 6 components should be configured.

The `server.xml` file associated with the default configuration is listed here. In the listing, advanced configuration components have been deleted (indicated by the ellipses, "…"), and comments have also been removed. The focus in this chapter is on the configuration of the remaining basic components. Chapter 6 covers configuration of the advanced components.

```
<Server port="8005" shutdown="SHUTDOWN">
...
  <Service name="Catalina">
    <Connector port="8080" protocol="HTTP/1.1"
               maxThreads="150" connectionTimeout="20000"
               redirectPort="8443" />
    <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
    <Engine name="Catalina" defaultHost="localhost">
      <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
             resourceName="UserDatabase"/>
      <Host name="localhost" appBase="webapps"
            unpackWARs="true" autoDeploy="true"
            xmlValidation="false" xmlNamespaceAware="false">
      </Host>
    </Engine>
  </Service>
</Server>
```

The nesting of components for this default configuration is shown graphically in Figure 5-1.

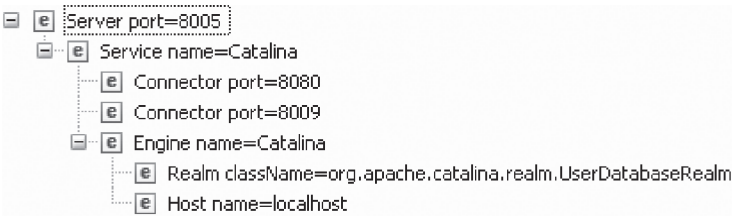

Figure 5-1: Nested configuration components in default server.xml

The next few sections examine each of these configurable components.

The Server Component

Our initial examination of the default `service.xml` file reveals that it configures a single service inside a single instance of the server component. In the `server.xml` file, the very first active line of the file defines the server component, which corresponds to the XML `<Server>` element. Here is the line from the configuration file:

```
<Server port="8005" shutdown="SHUTDOWN">
```

This tells Tomcat 6 to start a server instance (a JVM) listening to port 8005 for a shutdown command. Be careful if you need to change this port number. Tomcat or other servers may not start properly if identical ports are configured. The shutdown command will contain the text `SHUTDOWN`. This provides a graceful way for an administrator (or management console software) to shut down this Tomcat server instance. In a pinch, you can telnet to port 8005 and type in SHUTDOWN to take the server down; this cannot be done remotely for security reasons. Any unspecified attributes take on default values. The following table describes the allowed attributes of the `<Server>` element and their default values, followed by a list of the subelements that the XML `<server>` may contain.

Attribute	Description	Required?
`className`	The Java class of the server to use. This class is required to implement the `org.apache.catalina.Server` interface. By default, the Tomcat 6 code is `org.apache.catalina.core.StandardServer`.	No
`port`	The TCP port to listen to for the command specified by the shutdown attribute before shutting down gracefully. Tomcat will confirm that the connection is made from the same physical server machine. Together with a custom shutdown command string that you can specify (discussed next), this provides a measure of security against hacker attacks.	Yes
`shutdown`	The command text string that the server should monitor for, at the TCP port specified by the port attribute, before shutting down gracefully.	Yes

Within the `<Server>` element, the XML subelements shown in the following table are allowed.

Subelement	Description	How Many?
`<Service>`	A grouping of Connectors associated with an Engine. The Connectors handle different client protocols (HTTP, HTTPS, AJP, and so on) and manage request concurrency, while the Engine processes the requests.	1 or more
`<Listener>`	Lifecycle listener for interception of the server's lifecycle events (start, stop, before start, before stop, after start, after stop). The installed listener is called at a prescribed point of the server's lifecycle. Lifecycle events can be used by developers to add custom components, additional logging, management, resource allocation, or other added functionality to the server instance. See Chapter 6 for more details on the configuration of this advanced component.	0 or more
`<GlobalNamingResources>`	JNDI resources that are defined to be globally available throughout this server component instance. See Chapter 6 for more details on the configuration of this advanced component.	0 or more

The Service Component

The next line in the file defines a Service component. The main purpose of a Service component is to group a request-processing Engine with its configured set of protocol/concurrency handling Connectors.

The Service component is a top-level element, and is used to group together all the Connectors that may be used with the Catalina request-processing Engine. In the default `server.xml` file, the `<Service>` element is shown here:

```
<Service name="Catalina">
```

Here, a service instance was defined with the name `Catalina`. This name will be visible in the log file directory structure, clearly identifying the component. It is also used as the name to identify the service instance when using third-party or Web application–based management or administration tools.

A `<Service>` element can have the attributes shown in the following table.

Attribute	Description	Required?
`className`	The Java class name for the service class to use. By default, the Tomcat 6–supplied Catalina container code `org.apache.catalina.core.StandardService` is used. The default is adequate unless you're modifying Tomcat's source code.	No
`name`	A name for the service, used in logging, administration, and management. If you have more than one `<Service>` element inside the `<Server>` element, you must make sure their name attributes are different.	Yes

The subelements that a `<Service>` element can have are shown in the following table.

Subelement	Description	How Many?
`Connector`	This is a nested component that handles external client connections and feeds them to the Engine for processing. A Connector also manages the number of threads and their allocation for request handling. The configuration of Connectors is explained in detail in the next section.	1 or more
`Engine`	This is the request-processing component in Tomcat: Catalina.	Exactly 1

The Connector Component

It is necessary to understand the two modes of Tomcat operations before you can appreciate the role of the Connector component. Following are two very different ways of operating Tomcat 6:

- **Tomcat as an application server:** In this configuration, a Web server frontend (Apache, IIS, and so on) serves static content to end users, while all JSP and servlet requests are routed to the Tomcat server(s) for processing. In turn, Tomcat-hosted Web applications interface to backend J2EE-compliant services. (See Chapters 11 and 12 for an in-depth examination of this mode of Tomcat operation.)
- **Tomcat in standalone mode:** In this case, any static pages and graphic files from your Web application are served directly from the Tomcat 6 server. In this mode, an additional Web server frontend is not necessary because Tomcat is acting as both the Web server and the JSP/Servlet container. Tomcat 6 uses its built-in HTTP Connector to process the incoming HTTP request, bypassing the need for an external Web server altogether. Tomcat-hosted Web applications can interface to backend J2EE services.

The standalone mode of operation is less often used in public-facing production because of the huge gap in industry experience and security proofing between production Web servers (such as Apache and IIS) and Tomcat's built-in Web server. This mode is very easy to set up and manage because it does not involve any external Web server. The Tomcat 6 standalone mode of operation manages to achieve some impressive performance in working with a mix of both static and JSP/servlet-based requests, and is more than adequate for a deployment scenario with no public exposure and associated security risk. See Chapter 10 for an in-depth exploration of this mode of Tomcat operation.

Operating Tomcat in Application Server Configuration

In the application server configuration, some intelligent piece of software must run inside the Web server and decide on the requests that will be routed to the Tomcat server for processing. This usually exists in the form of a loadable module (that is, `mod_jk` or `mod_proxy` in Apache 2.2) containing a redirector plug-in.

In this case, multiple independent Tomcat servers may be running simultaneously (that is, across a networked bank of machines for scalability and load balancing), and the loadable module or redirector plug-in may also decide to which Tomcat server instance requests are sent. This sort of hardware configuration is technically known as a *Tomcat cluster*. (See Chapter 17 for a detailed description of clustering with Tomcat 6.)

For operational efficiency, the protocol between the Web server and the Tomcat instance(s) is not HTTP. It is a specially designed protocol called AJP. Chapter 4 provided you with an overview of the AJP protocol. You don't need to understand this protocol, but just need to be aware that there must be a native code extension to the Web server that routes incoming requests to Tomcat, and a corresponding piece of request-receiving software (a Connector component) at the Tomcat server-side that understands this protocol and connection convention.

In the default `server.xml` file, an HTTP 1.1 Connector is also defined for the Catalina service:

```
<Connector port="8080" protocol="HTTP/1.1"
        maxThreads="150" connectionTimeout="20000"
        redirectPort="8443" />
```

The following table describes two standard Connectors supplied with Tomcat 6.

Connector Name	Description
HTTP/1.1	Connects browser or Web services to the Catalina Engine using HTTP 1.1 if supported by the client, and adaptively falls back to using HTTP 1.0 if necessary. This Connector can also be configured to support secured HTTPS/SSL connections.
AJP/1.3	Used for connecting between external Web servers (Apache included) and Tomcat 6 using the AJP 1.3 protocol. It uses the external Web server for static Web content, while Tomcat 6 will handle servlet and JSP processing. It can also use the Web server's SSL support. See Chapters 11 and 12 for more details.

In the default `server.xml` file, you can see an additional configuration for the AJP Connector, supporting the AJP 1.3 protocol:

```
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

The Connector code in Tomcat 6 unifies the support of HTTP 1.0, HTTP 1.1, and HTTPS/SSL into a single Connector. AJP protocol support is available as a separate Connector. These Connectors are designed from scratch to work efficiently with Tomcat 6's optimized architecture. The combined protocol handling in one Connector can greatly simplify configuration, administration, management, and operation of Tomcat Connectors in general. Chapters 10–12 provide the details of configuring Connectors.

While you can have as many Connectors as you need in a service (to handle the different incoming client protocol requirements), there can be only one Engine. The Engine component is a container; think of it as Catalina — the Servlet/JSP processor. An Engine executes Web applications while processing incoming requests and generating outgoing responses.

The Engine Component

The one and only Engine component associated with the *Catalina* service is defined next in the default `server.xml` file:

```
<Engine name="Catalina" defaultHost="localhost">
...
</Engine>
```

An Engine is a container (see Chapter 4 for an architectural discussion of containers and nested elements), essentially representing a running instance of the servlet processor. The name `Catalina` is given to this configured Engine instance. An Engine can process a request destined for multiple configured virtual hosts. The `defaultHost` attribute indicates the virtual host to which Tomcat will direct a request if the request is not specifically destined for one of the virtual hosts configured in the `server.xml` file.

The attributes that an `<Engine>` element can have are shown in the following table.

Attribute	Description	Required?
`className`	The Java class name for the Engine code. If not specified, the default Tomcat code `org.apache.catalina.core.StandardEngine` is used, and is seldom overridden unless you're modifying Tomcat server code.	No
`backgroundProcessDelay`	The delay in seconds before the background processing is enabled for this Engine and other nested Host and Context components. Any nested component with its own `backgroundProcessDelay` set to a non-negative value will be ignored, indicating that it will manage its own background processing (if any). Background processing is typically used by components to perform low-priority tasks such as lazy reclamation of unused resources. One example of background processing is the occasional checking for Web application changes by a `<Host>` component for hot application redeployment. See Chapter 4 for more details on background processing resources. The default delay is 10 seconds.	No

Attribute	Description	Required?
`defaultHost`	Selects one of the virtual hosts within this Engine to process all the incoming requests by default. This is used only if the Engine cannot find the host named on the request within this `server.xml` file.	Yes
`jvmRoute`	This is an identifier used in load-balancing Tomcat 6. See Chapter 17 for more information on using this attribute and configuring Tomcat 6 for clustering and load balancing. Sticky session support relies on matching this identifier as it is part of the incoming request's session ID.	No
`name`	A name given to this Engine, which will be used in logging and by management applications.	Yes

As a container, the `<Engine>` element can have the subelements shown in the following table.

Subelement	Description	How Many?
`Host`	Each `<Host>` element specifies a virtual host handled by the Engine. Tomcat 6 can handle multiple virtual hosts per Engine/Service instance. This mirrors one of the most popular features of the Apache Web server.	1 or more
`Context`	Creates a Context (collection of settings for configurable properties/elements) for the Web applications that are automatically deployed when Tomcat 6 starts. The properties specified in this default Context are also available to all Web applications running within the Engine.	0 or 1
`Realm`	This Realm is used by default in the declarative security support (see Chapter 14 for more details) to map users into roles; it is used for authentication purposes. Each individual virtual host's `<Host>` and `<Context>` elements may have their own Realm for this purpose. If they do not define their own, the Realm configured at the Engine level is used.	0 or 1
`Valve`	Valves add processing logic into the request- and response-handling pipeline at the Engine level. Standard Valves are used to perform access logging, request filtering, implement single sign-on, and so on. Chapter 6 discusses the configuration of these standard Valves, as well as advanced configuration.	0 or more
`Listener`	This is used to configure lifecycle listeners that monitor the starting and stopping of the Engine. See Chapter 4 for information about how lifecycle listeners fit into Tomcat 6's architecture.	0 or more

The Realm Component

In the default `server.xml` file, after the configuration of the Engine component, the next configured nested component inside the Engine is a Realm component:

```
        <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
               resourceName="UserDatabase"/>
```

This configures a `UserDatabase` Realm to load the `tomcat-users.xml` file into memory for use in authentication by default applications such as the `manager` application. Chapter 14 covers the attributes for the `<Realm>` element, including how to specify your own XML file or data source for user authentication information.

A Realm is a security mechanism used to perform authentication and implement container-managed security. Essentially, Realms are data sources that provide mappings between usernames and passwords (for authentication), and between usernames and roles that users assume (for container-managed security). For example, user `johnf` may have password `xyzzy` (authentication) and a role of `supervisor`.

A Realm can access data sources external to Tomcat 6 where the user/password/role relationships are stored. There are many different implementations of Realms, differing only in the source from which they retrieve the information. Following are several types of Realms that are standard with Tomcat 6:

- **Memory:** Uses a memory-based table that is populated with the user/password/role mappings. Typically, this is read into memory from an XML file during server startup and stays static throughout the lifetime of the server. For the default implementation that comes with Tomcat, the size of the mappings is seriously constrained by the memory available. This is typically used only in testing and development, and seldom in production.
- **UserDatabase:** Implements a completely updateable and persistent memory Realm. It is backwardly compatible with the standard Memory Realm. Chapter 14 provides extensive coverage of UserDatabase.
- **JDBC:** Uses a relational database source for obtaining authentication information. Any other data sources with a JDBC-compatible access interface may also be used (for example, ODBC-compliant sources via the JDBC-to-ODBC bridge).
- **DataSource:** Similar to the JDBC realm, uses JDBC connections to obtain authentication information from a relational database source. However, it configures a lookup-based interface via JNDI when obtaining a JDBC data source for relational database access. This enables the pooling of multiple JDBC connections by the JNDI provider — providing more efficient access to the relational database when a Web application performs many authentications at the same time.
- **JNDI:** Uses Java Naming and Directory Interface (JNDI) to access the Realm data. This data is typically stored in an LDAP-based directory, although any authentication system compatible with the LDAP protocol can be used. (For instance, OpenLDAP, Microsoft, or Novell all have LDAP-compatible access drivers.)
- **JAAS:** Works in conjunction with the Java Authentication and Authorization Service (JAAS) to obtaining the authentication and authorization information for the Realm.

As mentioned earlier, Chapter 14 provides details about how to configure different Realms.

The Host Component

After the global UserDatabase Realm component configuration, the next configured component is a Host component. A Host component is a container; it can contain other nested components. The Host component represents a virtual host handled by a Tomcat 6 server instance. A single Tomcat 6 server can contain many virtual hosts. Each virtual host can be considered as a distinct request processing destination. A virtual host is configured as a `<Host>` element within the `server.xml` file. Each `<Host>` element defined within the enclosing `<Engine>` element represents another virtual host that is handled by this Engine. In our case, the Host definition is as follows:

```
<Host name="localhost" appBase="webapps"
      unpackWARs="true" autoDeploy="true"
      xmlValidation="false" xmlNamespaceAware="false">
```

This defines a virtual host named `localhost` matching the `defaultHost` specified in the `<Engine>` outer container. The applications to be deployed for this virtual host are located under the `$CATALINA_HOME/webapps` directory (all the examples from the Tomcat 6 distribution are installed there). In addition, the `unpackWARs` attribute specifies that if Tomcat 6 finds any WAR files in the `appBase` directory, they will be expanded before the Web application is executed. If you set `unpackWARs` to `false`, Tomcat executes the Web applications in place, without unarchiving them — saving space but sacrificing performance. The `autoDeploy` attribute is set to `true`, meaning that Tomcat actively scans for the addition of new Web applications or changes in existing ones, and then automatically deploys, or redeploys, them. See the description of the `autoDeploy` attribute in the next section for more details.

Chapter 15 discusses the techniques used to support virtual hosting. For now, however, Figure 5-2 illustrates the basic concept of virtual hosting.

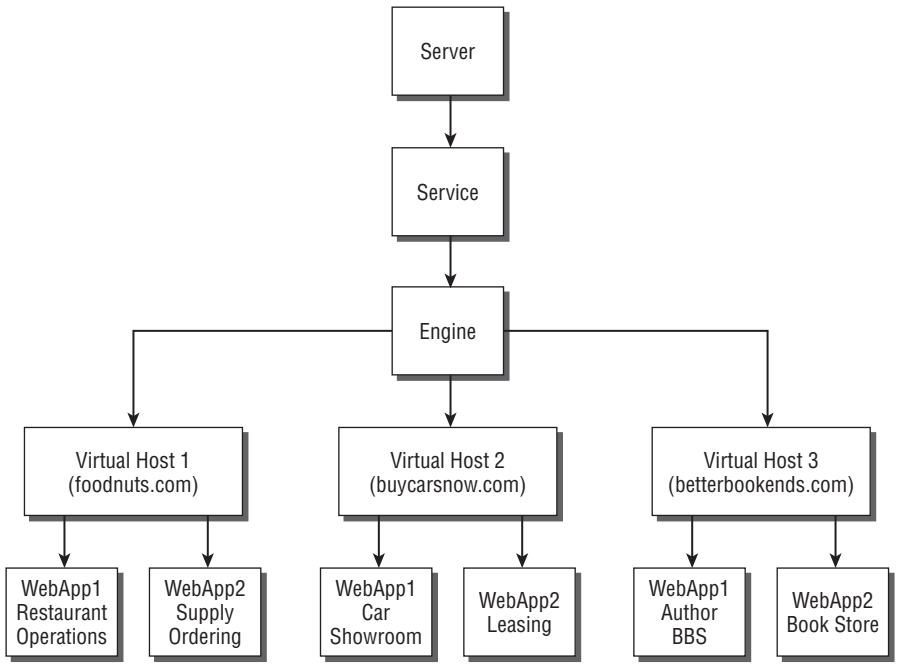

Figure 5-2: Virtual hosting in Tomcat 6

In this figure, a single Engine supports three fictitious Web sites via virtual hosts. The first one is `foodnuts.com`, the second one is `buycarsnow.com`, and the third one is `betterbookends.com`. Each virtual host is running a completely different Web application. The Engine is responsible for forwarding any incoming requests to the corresponding host. If the system were to be configured as depicted in the figure, there would be three `<Host>` elements nested within the single `<Engine>` definition.

A `<Host>` element is a container. It can have any one of the attributes shown in the following table.

Attribute	Description	Required?
`className`	The Java class that is used to handle requests for the host. The default is the Tomcat-supplied class `org.apache.catalina.core.StandardHost`, and this almost never needs to be changed.	No
`appBase`	Used to set the default application-deployment source directory. Tomcat 6 will look in this directory for applications to be deployed. The path should be specified relative to the installation or per-instance base directory for the Tomcat 6 server.	Yes
`autoDeploy`	Setting this attribute to `true` means that Web applications will be automatically deployed or redeployed while Tomcat 6 is running. This includes any new applications placed into the directory specified by `appBase` (in WAR form or unarchived), any application whose `web.xml` deployment descriptor has been modified, and any application whose Context descriptor has been modified. The default value is `true`. Background processing must be enabled for this to work properly. See `deployOnStartup` for auto application deployment during Tomcat startup.	No
`name`	The resolvable name of this virtual host.	Yes
`backgroundProcessDelay`	The delay in seconds before the background processing thread is enabled for this host and other nested components. Any nested component with its own `backgroundProcessDelay` set to a non-negative value will be ignored, indicating that it will manage its own background processing (if any). The default delay is `-1`, indicating that the parent's background processing thread should be used.	No
`deployOnStartup`	When set to `true`, automatically deploys Web applications from this host during component startup. The default is `true`.	No
`deployXML`	Used primarily in security sensitive installations to restrict the processing of `context.xml` embedded within a Web application (usually found in `META-INF/context.xml` of the WAR file). If this is set to `false`, the context configuration needs to be placed under	No

Attribute	Description	Required?
	`$CATALINA_HOME/conf/<engine>/<host>/<app>.xml`. The default value is `true`, allowing the processing of the `META-INF/context.xml` within any web applications deployed on the host.	
`errorReportValveClass`	Specifies the Java class that implements the error-reporting Valve used by this host. The default implementation is `org.apache.catalina.valves.ErrorReportValve`.	
`unpackWARs`	Set this to `false` if you want Tomcat 6 to run Web applications without unarchiving the WAR files found at the directory specified by the `appBase` attribute. The default is `true` and Tomcat 6 will unpack these applications. The tradeoff here is typically performance (lower performance when WAR files are not unarchived) versus storage (no need to write to the `appBase` directory).	No
`workdir`	Specifies a temporary working directory for servlets (and JSPs) that run within this host. These applications can get the temporary directory via a call to get the `javax.servlet.context.tempdir` property. If this is not specified, a directory under `$CATALINA_HOME/work` is used instead.	No

Note that there must be at least one `<Host>` entry associated with the `<Engine>` element. This makes sense because you must be able to reach the Engine by at least one virtual host name. This means the `defaultHost` attribute of the `<Engine>` element must be assigned with one of the `<Host>` entries.

The XML subelements that can be placed inside a `<Host>` element are described in the following table.

Subelement	Description	How Many?
`Context`	A `<Context>` can configure a set of property values for a Web application deployed within this host. There can be as many `<Context>` elements as there are Web applications. The default `server.xml` included with the Tomcat 6 distribution does not include any application Context. This enables a clean separation between server configuration and Web application configuration. Instead, all Web application Contexts can be maintained under the `$CATALINA_HOME/conf/<engine>/<host>` directory. In addition, an application can configure its own Context by including a `context.xml` file in the `META-INF` directory of the Web application WAR.	0 or more

Table continued on following page

Subelement	Description	How Many?
DefaultContext	The <DefaultContext> configures the set of property values for a Web application that is deployed within this host but that does not have its own <Context> specified. Typically, this <DefaultContext> is used by Web applications that are automatically deployed.	0 or 1
Realm	A Realm that can be accessed across all the Web applications running within this host, unless a lower-level component specifies its own Realm.	0 or 1

Web Application Context Definitions

In Tomcat 6, administrator-controlled application Context Descriptor XML files are placed in the $CATALINA_HOME/conf/<engine name>/<host name> directory. This is done to maximize the decoupling between server and application configuration, and to improve deployment security.

Web application–specific Context descriptors can be embedded in the WAR file, under META-INF/context.xml, and included with the Web application being deployed. Administrators can, at their discretion, disable the parsing of these embedded context.xml files on a per-host basis via the deployXML attribute of the host, documented earlier.

This concludes the examination of the default server.xml file. The remainder of the chapter examines the other configuration files found in the $CATALINA_HOME/conf directory.

The Default context.xml File

The context.xml in the $CATALINA_HOME/conf directory is the default Context descriptor loaded with every single application running on the Tomcat instance. Because of the global nature of this default Context, it should contain only configuration entries that you need to apply across all Web applications and across all virtual hosts. (The <DefaultContext> nested component has the same effect, but across all Web applications in a single host.)

This default context.xml contains only one line (all comments are removed in the following listing):

```
<Context>
    <WatchedResource>WEB-INF/web.xml</WatchedResource>
</Context>
```

The web.xml specified in the <WatchedResource> nested component is watched by the Context's background processing for changes. Whenever this resource is changed, the application is redeployed by the Tomcat autodeployer (unless this feature is disabled explicitly).

A <Context> element is a container. It can have any one of the attributes shown in the following table.

Attribute	Description	Required?
`className`	The Java class that is used to implement the per-application Context. The default is the Tomcat-supplied class `org.apache.catalina.core.StandardContext`, and does not need to be changed.	No
`allowLinking`	On Linux and other operating systems that support symbolic links, this allows files to be linked to resources outside the Web application tree. The default value is `false`. On Windows systems, which do not support symbolic links, make sure this value is `false`.	No
`antiJARLocking`	Uses specialized classloaders code to avoid locking up JAR files as much as possible while accessing class information inside the JAR file. Default is `false`. This should be turned on only if you experience a JAR file locking problem, such as failure in autodeployment.	No
`antiResourceLocking`	Uses specialized code to avoid file locking as much as possible. Default is `false`. This option should be left to default unless you experience file locking problems (typically during uninstall or redeployment of application).	No
`backgroundProcessDelay`	The delay in seconds before the background processing thread is enabled for this host and other nested components. Any nested component with its own `backgroundProcessDelay` set to a non-negative value will be ignored, indicating that it will manage its own background processing (if any). The default delay is `-1`, indicating that the parent's background processing thread should be used.	No
`cacheMaxSize`	Sets the maximum size for the resource cache, in KB. The default is `10240`.	No
`cacheTTL`	Time between attempts to revalidate items in cache, specified in milliseconds. The default is `5000`.	No
`cachingAllowed`	Determines if static resource (config files, images, Web pages, and so on) cache should be turned on. The default is `true`.	No
`caseSensitive`	Determines if case sensitivity checks should be performed. Default is `true`. Leave this on, especially on an operating system with case-insensitive file systems such as Windows.	No
`cookies`	Uses cookies to track session information from client access. Cookies are the most universally accepted means to perform session management. Default is `true`. If you turn this off, application-level URL rewriting is required to maintain session information.	No

Table continued on following page

Attribute	Description	Required?
crossContext	Enables cross-Context access when ServletContext.getContext() is called by a Web application. Default is false. You may need to turn this on for suites of Web applications that share information within the same virtual host.	No
docBase	The document base path of the Web application running in the Context. This is often called the Context root. If the application is deployed as a WAR file, the document base is, by default, set to webapps/<*name of war file*>. This path can be set relative to the appBase of the enclosing Host container, or specified as an absolute path.	No
override	Default is false. This indicates if settings in the local context.xml should be allowed to override settings in the DefaultContext specified with the master server.xml.	No
path	Specifies the context path of the Web application. If you want the application to be the default application for the specified host, use a path of "". In most cases, the default is sufficient. The default is to use the value specified in docBase, the WAR name, or the application Context file name.	No
privileged	Default is false. Should be set to true only if you are configuring a system application, such as the manager application.	No
processTlds	Specifies that TLDs should be processed when the Context starts up. Default is true. Typically no need to change this value.	No
reloadable	Default is false. Determines if Tomcat should monitor for changes in /WEB-INF/classes and /WEB-INF/lib for every application and reload when the application completes. This is typically used by developers during debugging.	No
swallowOutput	Default is false. Determines if any output to System.out or System.err should be captured and displayed in Tomcat's logs.	No
tldNamespaceAware	Determines if TLD processing and validation should be namespace-aware. Default is false. Turning this on helps in debugging TLDs, but adversely affects performance.	No
tldValidation	Determines if TLD validation should be performed during processing. Default is false. Turning this on helps in debugging TLDs but adversely affects performance.	No
unloadDelay	The number of milliseconds that Tomcat waits for Web applications to unload. The default is 2000.	No

Attribute	Description	Required?
unpackWAR	Default is true. When set to true, Tomcat unarchives the WAR file into the docBase directory before running it. If this is set to false, Tomcat attempts to run the WAR file without first unarchiving it.	No
useNaming	Default is true. Creates and provides to applications a Java EE JNDI–compatible InitialContext for resource lookup. This is required if a Web application uses database connections. See Chapter 13 for more information.	No
workdir	Specifies a temporary working directory for servlets (and JSPs) that run within this host. These applications can get the temporary directory via a call to get the javax.servlet.context.tempdir property. If this is not specified, a directory under $CATALINA_HOME/work is used instead.	No
wrapperClass	Specifies a wrapper class that implements the org.apache.catalina.Wrapper interface for wrapping servlets. The default implementation need not be changed in most cases, unless you are developing Tomcat server extensions.	No

Depending on the application, you may not have to craft a custom <Context> to get it deployed. Many applications are designed to run without the need for a custom Context descriptor.

The XML subelements that can be placed inside a <Context> element are described in the following table.

Subelement	Description	How Many?
Loader	Configures the class loader used to load classes for a Web application. See Chapter 9 for a thorough examination of class loaders used in Tomcat.	0 or 1
Manager	Configures the session manager for the context. The session manager creates, manages, and persists server-side sessions. There is typically no need to replace the default implementation unless you are implementing persistent sessions, failover, and/or clustering. Chapters 6 and 17 have more information on clustering configurations.	0 or 1
Realm	A Realm that can be accessed in the Web applications running within this Context.	0 or 1
Resources	The resource manager used to obtain resources. The default parent container implementation of org.apache.naming.resources.FileDirContext is sufficient.	0 or 1
WatchedResource	Tells the autodeployer to redeploy the Web application if any of the specified resources have changed.	0 or 1

This concludes the coverage of Context descriptor and the system default `context.xml` file in the `$CATALINA_HOME/conf` directory.

Authentication and the tomcat-users.xml File

Another configuration file found in the `$CATALINA_HOME/conf` directory is `tomcat-users.xml`. The `tomcat-users.xml` file is used by Tomcat 6 to authenticate `manager` tool users. Tomcat 6 makes use of a UserDatabase Realm component to accomplish this. The UserDatabase Realm enables modification of the loaded data and can properly persist (write back to the XML file) any changes made to the data. Only users assigned to role "manager" will be able to access the `manager` application.

The Default Deployment Descriptor — web.xml

According to the Servlet 2.5 specification, every Web application should include a deployment descriptor (`web.xml` file). This file must be placed in the `WEB-INF/` directory of the Web application and so is specific to just that Web application.

There is also a `web.xml` file under the `$CATALINA_HOME/conf` directory. This file is similar to a Web application's `web.xml` file. However, this particular `web.xml` file is used to specify the default properties for all Web applications that are running within this server instance. Be very careful when making modifications to this file (such as any additions or changes) because they affect all Web applications running on the same server instance. Note also that other application servers may or may not support a global default `web.xml`, as this is not a requirement for Servlet 2.5 standard compliance.

It is time to see what default server-wide properties are configured in this `web.xml` file. First, there is the standard XML header and a reference to the Servlet 2.4 schema (Tomcat 6 is Servlet 2.5–compliant, but as of this writing, the global `web.xml` uses the Servlet 2.4 schema). Unlike `server.xml`, `web.xml` can be formally validated against a schema:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
    version="2.4">
```

The Servlet 2.4 schema provides a significantly more rigorous mechanism for document validation than the DTD used in earlier specifications. Chapter 7 provides a detailed discussion of the Servlet 2.4 and 2.5 schemas.

Configuring the Default Servlet for Static Resources

In the following `<servlet>` definition, a default servlet is specified. This default servlet is used to serve any static resources (static HTML files, GIF files, and so on) within all Web applications, and also to serve directory listings (if it is enabled; see the following listing). In other words, this default servlet provides equivalent capability to a standard Web server. In fact, when using Tomcat in the standalone mode of operation, it is the default servlet that handles static resources.

The following segment in the `web.xml` file configures this default servlet:

```
<servlet>
  <servlet-name>default</servlet-name>
```

```
    <servlet-class>
       org.apache.catalina.servlets.DefaultServlet
    </servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
    </init-param>
    <init-param>
      <param-name>listings</param-name>
      <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
```

Default Directory Listings and Other Customizations

You can disable directory listings by setting the `listings` parameter to `false`. You can further customize the behavior of the default servlet by using the parameters shown in the following table. Only the frequently changed parameters are shown.

Parameter	Description
`listings`	Determines if directory listings should be shown when a URL reaches the root of a directory. This is implemented to provide compatible behavior with most Web servers. Set this to `false` if you want to tighten security, or if some directories are very large and take significant time to display.
`readonly`	Controls if write HTTP methods such as `PUT` and `POST` are allowed. Default is `true`. You need to override this if you need to enable upload.
`input`	The buffer size used when reading resources. By default it is set to 2048 or 2KB.
`Output`	The buffer size used when writing resources. By default it is set to 2048 or 2KB.
`globalXsltFilelocalXsltFile`	Specifies an XSLT stylesheet to use when displaying directory listings. This can be used to customize the look of the directory listings. `localXsltFile` specifies a file name in the directory that Tomcat looks for to format the listing. `globalXsltFile` specifies a stylesheet that can be located in any directory, via an absolute path. If a `localXsltFile` is not specified, the global stylesheet is used. If there are no global or local XSLT stylesheets, then the listings will not be formatted.
`sendfileSize`	Tomcat 6 supports the platform-specific use of a system-level asynchronous `sendfile()` to optimize the serving of large static resources (such as images). This parameter sets the threshold of file size before a file is considered for `sendfile()` transmission. The size is specified in KB and the default is 48.

The next section of the default `web.xml` configures the `invoker` servlet.

Configuring the Invoker Servlet

The `invoker` servlet can be used to load and execute any servlet directly, using a URL similar to the following:

```
http://<host name>/<context path>/servlet/<servlet name>
```

Because of its capability to invoke any servlet directly (with or without prior configuration within a Web application) the `invoker` servlet is considered a major security risk in production systems. Therefore, this servlet should be used only in test configurations. Tomcat 6's default `web.xml` file has the `invoker` servlet configuration commented out for this security-related reason. You can uncomment it and enable the servlet on test configurations.

The `invoker` servlet is configured as follows:

```
<!--
  <servlet>
    <servlet-name>invoker</servlet-name>
    <servlet-class>
      org.apache.catalina.servlets.InvokerServlet
    </servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>
-->
```

Configuring the JspServlet

The JspServlet converts JSP pages to servlets and executes them. This servlet has a development code name of Jasper but is frequently referred to as Japser. Jasper is used to process JSP pages. The following `web.xml` segment shows the configuration of this servlet:

```
<servlet>
        <servlet-name>jsp</servlet-name>
        <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
        <init-param>
            <param-name>fork</param-name>
            <param-value>false</param-value>
        </init-param>
        <init-param>
            <param-name>xpoweredBy</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>3</load-on-startup>
</servlet>
```

Other initialization parameters for Jasper that are used frequently are explained in the following table.

Parameter	Description
development	Default is set to true. Indicates that JSP is under development and that Jasper should honor the modificationTestInterval parameter to update/recompile JSPs if modified.
fork	Default is set to true. Indicates that a separate JVM should be forked to compile JSPs. These eliminate resource contention and potential class loader conflicts during JSP or Web application development.
checkInterval	The time interval, specified in seconds, between which Jasper will check to see if a JSP needs to be recompiled. Default is 0, indicating that checks are performed with background processing.
modificationTestInterval	Used mainly with development set to true. The threshold value, specified in seconds since the last compile, that a JSP is eligible for modification check again. Default is 4.
compiler	The buffer size used when reading resources. By default, it is set to 2048 or 2KB.
classdebuginfo	The buffer size used when writing resources. By default, it is set to 2048 or 2KB.
keepgenerated	Default is true. This causes Jasper to retain the compiled JSPs (servlets) between invocations. This can significantly speed up the application startup time.
mappedfilegenStrAsChar Array dumpSmaptrimSpaces supressSmap	These are optimization and debugging options for JSP developers. Your development team may request specific settings during the debugging phases of their projects.
scratchdir	Temporary directory required during JSP compilation. Default is a temporary directory under the $CATALINA_HOME/work directory.
xpoweredBy	Generates the X-Powered-By special header. Default is false.
compilerTargetVM	The target VM for the compiled servlets. Default is 1.4 on JDK 1.4 and 1.5 for higher versions of JDK.
compilerSourceVM	The source VM for the generated servlets. Default is 1.4 on JDK 1.4 and 1.5 for higher versions of JDK.

SSI and CGI Servlets Configuration

The next set of servlets is commented out. You should uncomment them if you plan to add Apache-style server-side include (SSI) processing features to the standalone Tomcat 6 server.

```
  <!--
    <servlet>
      <servlet-name>ssi</servlet-name>
      <servlet-class>org.apache.catalina.ssi.SSIServletServlet</servlet-class>
      <init-param>
        <param-name>buffered</param-name>
        <param-value>1</param-value>
```

(continued)

```
      </init-param>
      <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
      </init-param>
      <init-param>
        <param-name>expires</param-name>
        <param-value>666</param-value>
      </init-param>
      <init-param>
        <param-name>isVirtualWebappRelative</param-name>
        <param-value>0</param-value>
      </init-param>
      <load-on-startup>4</load-on-startup>
    </servlet>
  -->
```

The next servlet definition is also used exclusively for configuring the Tomcat 6 server to mimic an Apache Web server. If you would like the standalone Tomcat 6 server to process CGI, you need to uncomment the following section:

```
  <!--
    <servlet>
      <servlet-name>cgi</servlet-name>
      <servlet-class>org.apache.catalina.servlets.CGIServlet</servlet-class>
      <init-param>
        <param-name>clientInputTimeout</param-name>
        <param-value>100</param-value>
      </init-param>
      <init-param>
        <param-name>debug</param-name>
        <param-value>6</param-value>
      </init-param>
      <init-param>
        <param-name>cgiPathPrefix</param-name>
        <param-value>WEB-INF/cgi</param-value>
      </init-param>
       <load-on-startup>5</load-on-startup>
     </servlet>
  -->
```

Matching URLs: Servlet Mappings

A `<servlet-mapping>` element specifies how incoming requests containing a specific URL pattern are to be handled:

```
    <servlet-mapping>
      <servlet-name>default</servlet-name>
      <url-pattern>/</url-pattern>
    </servlet-mapping>
```

The rule set up here specifies: When you see a URL request fitting the pattern, route it to the default servlet.

For example, if the host is `www.wrox.com`, and a standalone version of the Tomcat 6 server is running, then the following URL will map to the servlet named default:

```
http://www.wrox.com/<context path>/
```

The `<context path>` portion defaults to the name of the application's WAR file if not explicitly specified in a `<Context>` element.

If you look back at the `<servlet>` definition earlier in this file, you see that it was specified that the `org.apache.catalina.servlets.DefaultServlet` will be handling this request.

The second `<servlet-mapping>` is commented out because it is for the security-sensitive `invoker` servlet. You may uncomment this to enable the `invoker` servlet in test configurations:

```
<!--
  <servlet-mapping>
    <servlet-name>invoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
-->
```

The rule here specifies the following: When you see a URL request fitting the pattern `/servlet/*`, route it to the `invoker` servlet.

Therefore, the following URL request is sent to a servlet called `invoker`:

```
http://www.wrox.com/<context path>/servlet/<name of servlet>
```

If you refer back in the file, the `org.apache.catalina.servlets.InvokerServlet` is specified to process the request. This `invoker` servlet in turn invokes the servlet that is named by examining the incoming URL.

The next two `<servlet-mapping>` elements specify that all URLs containing `*.jsp` and `*.jspx` should be passed to the servlet named `jsp` for processing. In the earlier `<server-mapping>`, the `jsp` servlet is declared to be the `org.apache.jasper.servlet.JspServlet` class:

```
<servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
<servlet-mapping>
      <servlet-name>jsp</servlet-name>
      <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

How server.xml, Context Descriptors, and web.xml Work Together

Figure 5-3 illustrates how an incoming URL is parsed by the various components of a Tomcat 6 server, and how a `<servlet-mapping>` with a `<url-pattern>` controls the final mapping of the request to a specific servlet in a Web application.

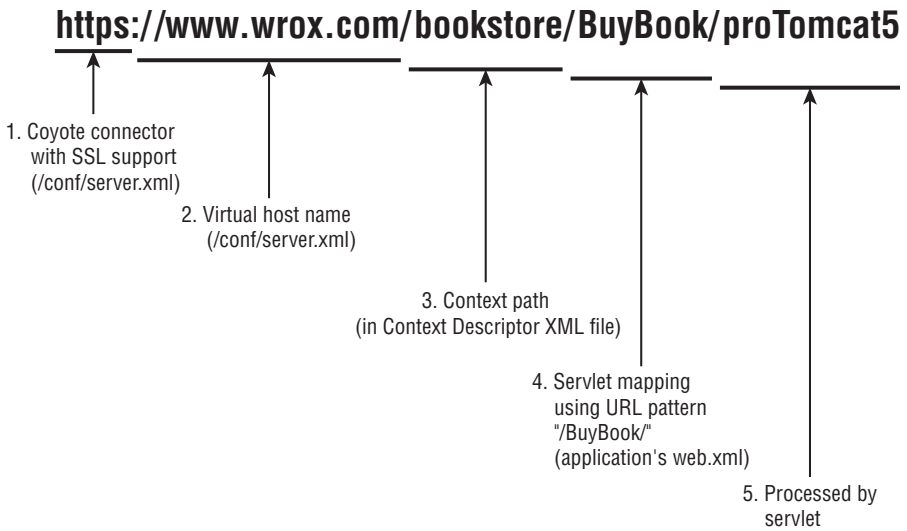
https://www.wrox.com/bookstore/BuyBook/proTomcat5
1. Coyote connector with SSL support (/conf/server.xml)
2. Virtual host name (/conf/server.xml)
3. Context path (in Context Descriptor XML file)
4. Servlet mapping using URL pattern "/BuyBook/" (application's web.xml)
5. Processed by servlet

Figure 5-3: How server.xml and web.xml are involved in URL parsing

In the figure, the URL `https://www.wrox.com/bookstore/BuyBook/proTomcat5` is parsed through the nested components that make up a Tomcat server. First, the protocol portion (`https://`) is parsed by the *Service* and the Coyote Connector with SSL support is selected, and the request is passed to the *Engine*. Next, the host name (`www.wrox.com`) is parsed by the Engine and one of its *Host* components is selected (the one that matches the `www.wrox.com` host name).

The Host then attempts to match the URL against the Contexts of its deployed Web applications — the match in this case is `/bookstore`, and the `bookstore` Web application is selected to handle the request (the Context information itself is stored in a Context descriptor file). Last but not least, the *Context* hosting the Web application performs a match against the `<servlet-mapping>` defined in the deployment descriptor (the `web.xml` file of the Web application), and the URL pattern `/BuyBook/*` matches the `BookPurchase` servlet. This servlet is finally handed the URL request to process. It is easy to see how the component hierarchy helps in forwarding the request to a single servlet in a Web application for processing.

SSI and CGI Mappings

Now it's time to take a look at the next section of the default `web.xml` file.

The next two default servlet mappings are commented out. They support SSI and CGI when Tomcat 6 is configured to work in standalone mode:

```
<!--
  <servlet-mapping>
    <servlet-name>ssi</servlet-name>
    <url-pattern>*.shtml</url-pattern>
  </servlet-mapping>
-->
  <!-- The mapping for the CGI Gateway servlet -->
<!--
  <servlet-mapping>
```

```
    <servlet-name>cgi</servlet-name>
    <url-pattern>/cgi-bin/*</url-pattern>
  </servlet-mapping>
-->
```

Session Timeout Configuration

The `<session-config>` element configures the amount of time during which Tomcat 6 will maintain a session on the server side on behalf of a client. For example, the client may be in the middle of an online shopping transaction and still have products in the shopping cart. In this case, if the client does not return to the cart for 30 minutes, and no session persistence is used (see Chapter 6 for a description of the session persistence manager), all the client's cart information is lost. As administrators, it is important to balance carefully the `<session-timeout>` value with the potential of overloading the server with too many stale sessions:

```
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

Handling Client-Side Helper Activation: Mime Mappings

The next set of elements contains the default `<mime-mapping>` elements. Tomcat 6 uses these mappings to serve static files with specific extensions to the client. It generates an HTTP `Content-Type` header when transmitting the file to the client (typically a browser). Most browsers use a helper application to process the file being transmitted if they recognize the content type specified. For example, Microsoft Internet Explorer may start Microsoft MediaPlayer when it detects the `video/x-mpeg` content type. Note that these are only the default mappings; a Web application's own deployment descriptor (`web.xml` file) can override or add to this list:

```
    <mime-mapping>
      <extension>abs</extension>
      <mime-type>audio/x-mpeg</mime-type>
    </mime-mapping>
    <mime-mapping>
      <extension>ai</extension>
      <mime-type>application/postscript</mime-type>
    </mime-mapping>
    <mime-mapping>
      <extension>aif</extension>
      <mime-type>audio/x-aiff</mime-type>
    </mime-mapping>
    <mime-mapping>
      <extension>aifc</extension>
      <mime-type>audio/x-aiff</mime-type>
    </mime-mapping>
      ... more mime mappings...
    <mime-mapping>
      <extension>Z</extension>
      <mime-type>application/x-compress</mime-type>
    </mime-mapping>
    <mime-mapping>
      <extension>z</extension>
      <mime-type>application/x-compress</mime-type>
```

(continued)

```
    </mime-mapping>
    <mime-mapping>
      <extension>zip</extension>
      <mime-type>application/zip</mime-type>
    </mime-mapping>
```

Simulating Apache Web Server: Welcome File Handling

The last section in the `web.xml` file pertains only to Tomcat's standalone mode of operation. To be compatible with the default behavior of the Apache Web server, the default servlet will display a welcome file if the incoming URI is terminated, as shown in the following example:

```
http://www.wrox.com/
```

The default servlet examines the root directory of the named virtual host (`www.wrox.com`) and looks for `index.html`, `index.htm`, or `index.jsp` in turn to be displayed. Each Web application may override this list in its own deployment descriptor (`web.xml`) file:

```
    <welcome-file-list>
      <welcome-file>index.html</welcome-file>
      <welcome-file>index.htm</welcome-file>
      <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
  </web-app>
```

In the following section, another file in the `$CATALINA_HOME/conf` directory — `catlina.policy` — is examined.

Fine-Grained Access Control: catalina.policy

Chapter 14 provides complete coverage of the role of the Tomcat security manager and its use of this policy file. For now, it is adequate to take a quick browse through the file to understand how it provides access control for a Tomcat 6 server administrator.

Tomcat 6 leverages the built-in fine-grained security model of Java 2. When enabled, the basis of the security system is as follows:

> Any access to system resources that is not explicitly allowed is prohibited.

This means that you must anticipate all the resources that the Tomcat 6 server will access, and explicitly grant permission for it to do so.

By default, Catalina starts up without security. You need to start Tomcat 6 with the following option for it to run with a security manager:

```
> startup -security
```

It is only in this secured mode that the `catalina.policy` file will be read, processed, and enforced. Some of the more important sections of the `catalina.policy` file are discussed later in the chapter, but details of the file are not covered at this time. The general policy entry is in the following form, where the `<security principal>` is typically a body of trusted code:

```
grant <security principal> { permission list... };
```

If you look at the `catalina.policy` file, you see that the first set of permissions grant code from the Java compiler directories all access to all resources (this is essentially the Java compiler and runtime system code):

```
// These permissions apply to javac
grant codeBase "file:${java.home}/lib/-" {
        permission java.security.AllPermission;
};
// These permissions apply to all shared system extensions
grant codeBase "file:${java.home}/jre/lib/ext/-" {
        permission java.security.AllPermission;
};
// These permissions apply to javac when ${java.home] points at $JAVA_HOME/jre
grant codeBase "file:${java.home}/../lib/-" {
        permission java.security.AllPermission;
};
// These permissions apply to all shared system extensions when
// ${java.home} points at $JAVA_HOME/jre
grant codeBase "file:${java.home}/lib/ext/-" {
        permission java.security.AllPermission;
};
```

One clear message here is that you must protect these directories using your operating system file-protection features (that is, via file ownership and permission settings).

The next section grants Catalina server code, logging code, and API libraries access to all resources:

```
// These permissions apply to the daemon code
grant codeBase "file:${catalina.home}/bin/commons-daemon.jar" {
        permission java.security.AllPermission;
};
// These permissions apply to the logging API
grant codeBase "file:${catalina.home}/bin/tomcat-juli.jar" {
        permission java.security.AllPermission;
};
// These permissions apply to the server startup code
grant codeBase "file:${catalina.home}/bin/bootstrap.jar" {
        permission java.security.AllPermission;
};
// These permissions apply to the servlet API classes
// and those that are shared across all class loaders
// located in the "lib" directory
grant codeBase "file:${catalina.home}/lib/-" {
        permission java.security.AllPermission;
};
```

Again, in a secure configuration, you must be careful to lock down the preceding directories, thus preventing an attacker from adding malicious code to them. Any class files introduced into these directories will automatically be granted access to all system resources.

The final set contains the permissions given to Web applications by default. They are significantly more restrictive (that is, they are never granted the all-powerful permission `java.security.AllPermission`).

The first section enables access to system properties that enable JNDI and JDBC access:

```
grant {
        // Required for JNDI lookup of named JDBC DataSource's and
        // javamail named MimePart DataSource used to send mail
        permission java.util.PropertyPermission "java.home", "read";
        permission java.util.PropertyPermission "java.naming.*", "read";
        permission java.util.PropertyPermission "javax.sql.*", "read";
```

The next section enables read-only access to some operating system description properties (the type of operating system that is running and what it uses to separate file extensions in a file name):

```
        // OS Specific properties to allow read access
        permission java.util.PropertyPermission "os.name", "read";
        permission java.util.PropertyPermission "os.version", "read";
        permission java.util.PropertyPermission "os.arch", "read";
        permission java.util.PropertyPermission "file.separator", "read";
        permission java.util.PropertyPermission "path.separator", "read";
        permission java.util.PropertyPermission "line.separator", "read";
```

The third section enables read-only access to some JVM-specific properties that are often used in application programming:

```
        // JVM properties to allow read access
        permission java.util.PropertyPermission "java.version", "read";
        permission java.util.PropertyPermission "java.vendor", "read";
        permission java.util.PropertyPermission "java.vendor.url", "read";
        permission java.util.PropertyPermission "java.class.version", "read";
        permission java.util.PropertyPermission "java.specification.version",
                                              "read";
        permission java.util.PropertyPermission "java.specification.vendor",
                                              "read";
        permission java.util.PropertyPermission "java.specification.name", "read";
        permission java.util.PropertyPermission "java.vm.specification.version",
                                               "read";
        permission java.util.PropertyPermission "java.vm.specification.vendor",
                                              "read";
        permission java.util.PropertyPermission "java.vm.specification.name",
                                              "read";
        permission java.util.PropertyPermission "java.vm.version", "read";
        permission java.util.PropertyPermission "java.vm.vendor", "read";
        permission java.util.PropertyPermission "java.vm.name", "read";
```

The next section is required for the use of MX4J (formerly called OpenJMX), providing JMX support for Tomcat 6 (see Chapter 16 for more information on JMX):

```
        // Required for OpenJMX
        permission java.lang.RuntimePermission "getAttribute";
```

The last two sections provide access to XML parser debug and precompiled JSPs, required frequently during code development (see JavaBean and JAXP specifications for more details on these properties):

```
       // Allow read of JAXP compliant XML parser debug
       permission java.util.PropertyPermission "jaxp.debug", "read";
```

```
        // Precompiled JSPs need access to this package.
        permission java.lang.RuntimePermission
           "accessClassInPackage.org.apache.jasper.runtime";
        permission java.lang.RuntimePermission
           "accessClassInPackage.org.apache.jasper.runtime.*";};
```

These minimal permissions are granted by default to Web applications. A typical secured production configuration will require opening up additional access to the Web applications, such as socket access to a JDBC server or network access to an external authentication system.

catalina.properties: Finer-Grained Control over Access Checks

Last but not least, the `$CATALINA_HOME/conf/catalina.properties` file is read during a secured Tomcat 6 server startup, and allows administrators to configure access control at a Java package level. This level of restriction causes a `SecurityException` to be reported should an errant or malicious Web application attempt to access these Tomcat 6 internal classes directly, or if a new class definition is attempted under these highly privileged packages.

The following lines in the `catalina.properties` file specify the name of the internal packages that should be restricted. Where partial package names are specified, any subpackages are protected as well.

```
package.access=sun.,org.apache.catalina.,org.apache.coyote.,org.apache.tomcat.,
org.apache.jasper.,sun.beans.
package.definition=sun.,java.,org.apache.catalina.,org.apache.coyote.,
org.apache.tomcat.,org.apache.jasper.
```

Other lines in `catalina.properties` include path definitions for the `common`, `server`, and `shared` class loaders (see Chapter 9 on Tomcat class loading):

```
common.loader=${catalina.home}/lib,${catalina.home}/lib/*.jar
server.loader=
shared.loader=
```

The last line of `catalina.properties` enables the use of the `String` cache. This cache is used internally in the conversion methods `ByteChunk.toString()` and `CharChunk.toString()`.

```
tomcat.util.buf.StringCache.byte.enabled=true
```

Bootstrapping Configuration

Before concluding this chapter, it is important to reiterate that any Web-based administration tool (such as the `admin` tool discussed in the last section of this chapter) is itself a Tomcat 6–hosted Web application. This means that Tomcat 6 must be running and operating for you to be able to access it. Tomcat 6 includes a default bootstrap configuration to ensure that these tools can start under most circumstances. However, in the unlikely event that you have made manual modifications and the Tomcat 6 instance will not start up, your only remaining recourse is to edit the XML configuration files via a text editor. This is the primary reason why this chapter has devoted considerable coverage to Tomcat administration from the perspective of manually editing the XML configuration files.

A Final Word on Differentiating Between Configuration and Management

Inexperienced Tomcat administrators often confuse the Web-based configurator (`admin`) application with the Web-based manager (the `manager` Web application, covered in more detail in Chapter 8) application. At first glance, they appear to offer similar capabilities. In reality, however, they are completely separate Web applications that offer a mutually exclusive set of administrative capabilities. One easy way to distinguish between the two is to realize that the Web-based configurator is used primarily to modify static configuration files that will be read and used by Tomcat *before server startup*, and that the `manager` application is used to manage Tomcat operations *after server startup*. In other words, `admin` is used for configuration, and `manager` is used during operations.

Tomcat 6 Web-Based GUI Configurator

Tomcat 5.5 and earlier versions had a Web-based administration tool called the admin Web application. This `admin` application enabled management of the Tomcat server itself, including the capability to add, delete, or modify `Connectors`, `Hosts`, and `Context`; manage `Resources` such as `DataSources` and `Environment` parameters; and manage users and roles. In short, it provided a Web-based GUI for tasks that otherwise would require editing Tomcat's configuration files (`server.xml, tomcat-users.xml`, and so on) and restarting the Tomcat server.

Unfortunately, at the time of this writing, this `admin` tool has not been ported to Tomcat 6. Even if it does get ported, it is disabled by default for security reasons and should never be enabled on a production Web site. Hence, knowledge of Tomcat's `server.xml` file and the ability to configure it manually is essential for a Tomcat administrator.

This section provides an overview of how to configure and start the admin tool. Check for online updates of this book for coverage of future releases of Tomcat 6 that may include the admin tool.

Figure 5-4 shows a typical screen from this tool. In this case, an HTTP Connector component source is being configured.

Manual editing of an XML configuration file is always supported by Tomcat, and you may wish to administer your server purely through this method. For those more comfortable with a GUI, the Web-based GUI provided by the admin tool provides an alternative. A major advantage of a Web-based GUI configuration is the capability to perform remote, off-site administration. Of course, for some, this may be viewed as a potential vulnerability. With the admin tool, administrators can reconfigure and maintain server instances wherever a Web browser connected to the Internet is available.

Even though the configuration is performed graphically, the XML configuration files are still being modified. These files are kept in the `$CATALINA_HOME/conf` directory of the Tomcat 6 distribution (or, if you have configured multiple Tomcat 6 instances, the corresponding `$CATALINA_BASE/conf` directory).

Figure 5-5 illustrates how Web-based Tomcat configuration can be performed, and the behind-the-scenes work that takes place.

In Figure 5-5, the user changes the value of a certain attribute of a Tomcat component via the Web-based GUI. The admin Web application then makes the corresponding change in the configuration XML file.

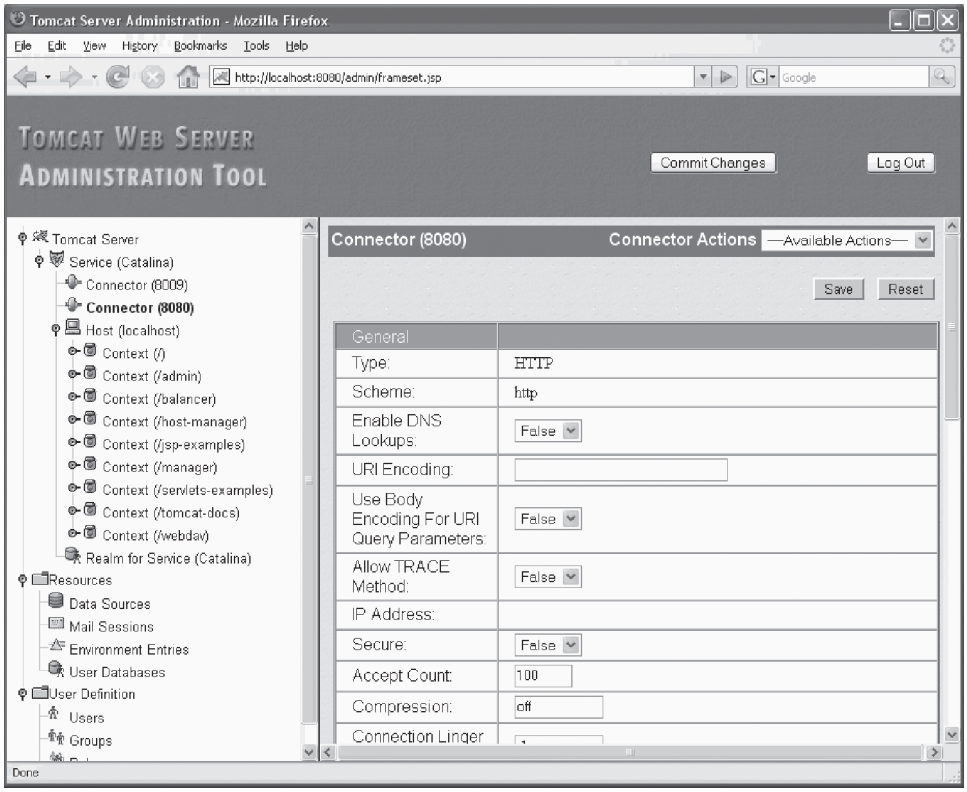

Figure 5-4: Tomcat's Web-based configurator (admin tool)

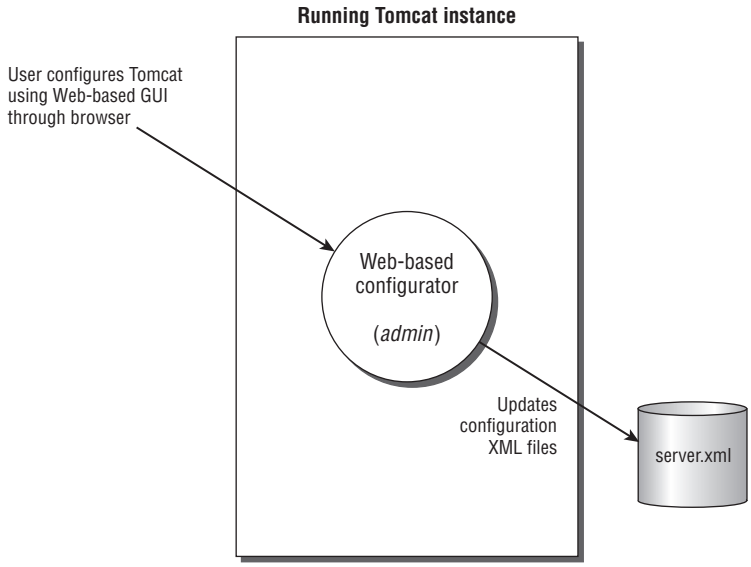

Figure 5-5: How Tomcat's Web-based configuration works

The user can control when the change to the XML file occurs by clicking the Commit Changes button on the top panel of the user interface. The admin Web application can be conceptually viewed as a user-friendly editor for the XML-based configuration files, whereby the Commit Changes button enables the user to save any changes to the files.

Summary

This chapter described the configuration of Tomcat 6 in detail, by examining the configuration of its architectural components via XML files. Because every configurable component maps to elements in XML files in the configuration directory, all the Tomcat server configuration files in the `$CATALINA_HOME/conf` directory of the Tomcat 6 distribution were covered. These files include the following:

- `server.xml`
- `context.xml`
- `tomcat-users.xml`
- `web.xml`
- `catalina.policy`
- `catalina.properties`

It is obvious from the discussion that `server.xml` is the essential server configuration file for Tomcat 6. To understand the model of configuration, it is necessary to understand the concept of a top-level component, a container hierarchy, and nested components. In addition, the function and configuration of the following Tomcat components were covered:

- Server
- Service
- Connector
- Engine
- Host
- Context
- Realm

You learned how these components relate to one another and work together during normal Tomcat 6 operation. To conclude this chapter, let's review some of its key points:

- The `tomcat-users.xml` file is the authentication and authorization data supply for a Memory Realm that is used by the Tomcat `manager` application, as well as a sample Realm implementation that programmers may use. In a production system, a more robust implementation of a Realm (such as a JDBC Realm or a JNDI Realm) should be used.
- The default `web.xml` file in `$CATALINA_HOME/conf` specifies properties that are used in every single Web application running on the server. Many of the default servlets configured here

provide Web server–like features (serving static content, SSI, CGI, and so on) for running Web applications.

- While Tomcat 6 starts up by default in an unsecured mode, the `catalina.policy` file is very important in secured Tomcat 6 installations. It specifies in excruciating detail what can be accessed by whom — and anything else that is not specified cannot be accessed. Tomcat 6 takes advantage of the sophisticated, built-in security infrastructure of Java 2. To protect against tampering with Tomcat internal classes, the `catalina.properties` file can be used to restrict internal package access and definition.
- The in-depth analysis of these configuration files should provide an understanding of the basic configuration features of the Tomcat 6 server.

Chapter 6 covers advanced Tomcat features, such as access log administration, single sign-on, and much more.

6

Advanced Tomcat Features

Earlier chapters discussed Tomcat 6 administration basics and Tomcat system architecture. This chapter explores a collection of administrative tasks that involve advanced features built into standard Tomcat 6. As a Tomcat 6 administrator, you are likely to encounter requests for many of these features from the development team.

More specifically, the following advanced administration tasks are explored:

- Administering access logs
- Working with single sign-on across Web applications
- Request filtering
- Installing a Persistent Session Manager
- Setting up Tomcat JNDI emulation resources to provide developers access to external JDBC and JNDI resources
- Setting up Tomcat for access to a JavaMail session
- Configuring lifecycle listeners

Note that configuration of Realms, a very important advanced Tomcat 6 administration topic, is discussed in Chapter 14. A basic understanding of Tomcat 6's security infrastructure is a prerequisite for appreciating the configuration options in Realms configuration.

This chapter serves as a "cookbook" for these specific tasks. For each task, we provide reasons why a user or a developer may need the feature, followed by the configuration and administrative details. Finally, we present a practical sample configuration, which you can experiment with. Useful hints, tips, or problems that may apply are pointed out along the way.

Valves — Interception Tomcat-Style

Valves are intrinsic architectural elements, similar to filters and specific to Tomcat, and have been available since early versions of Tomcat 4. As a filter-like element, a Valve can intercept any incoming request and outgoing response. In Tomcat 6, a set of standard Valves is delivered with the distribution. How do Valves differ from filters?

Architecturally, they are managed by the Engine and are given access to the incoming request from the Connectors before (and after) they are handled by the servlet and JSP processing logic. Logically, they can also be applied on a per-virtual-host or per-Web-application level (although application developers will typically use Tomcat 6 application filters instead of Valves if they need per-application filtering). Valves offer value-added functionality that includes the following:

- Access logging
- Single sign-on for all Web applications running under Tomcat
- Request filtering/blocking by IP address and/or host name
- Dumping of incoming and outgoing request headers for debugging purposes

The following sections examine the set of standard Valves available with Tomcat 6.

Standard Valves

Valves are *nested components* in the Tomcat 6 configuration component model (configured using the `<Valve>` XML element in the `server.xml` file) that can be placed inside `<Engine>`, `<Host>`, or `<Context>` containers (refer to Chapter 4 for architectural details on containers). The Engine passes an incoming request and outgoing response through any Valve that is incorporated within these containers. This process is illustrated in Figure 6-1.

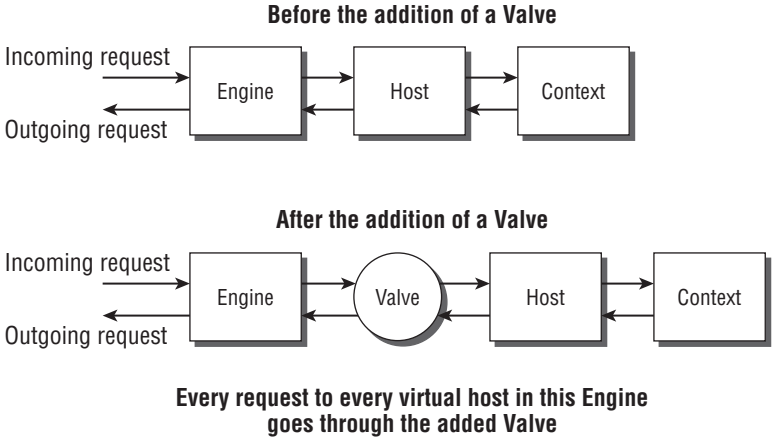

Figure 6-1: Position of Valves in Tomcat 6's architecture

In Figure 6-1, every single incoming request is passed through the added Valve. Because of this, Valves can be configured to perform work on each request. The Java language programming interface,

`org.apache.catalina.Valve`, is well-documented. Java application programmers may create their own Valves using this programming interface. For the most common application of Valves, however, Tomcat 6 already has basic implementations built in. The following table describes these standard Valves.

Valve Name	Description
Access Logging	Enables logging of the request (the URL of the resource requested, date and time of request)
Single Sign-on	Enhances the user experience by requesting a password only once, even when the user accesses different Web applications on the same host or server
Request Filtering	Enables selective filtering (blocking) of incoming requests based on a list of IP addresses or host names
Request Dump	Prints the headers and cookies of incoming requests and outgoing responses to a log

The configuration of each of these Valves is illustrated in the following sections.

> *Note that internally, Tomcat 6 also utilizes Valves to perform many of its standard actions. For example, Tomcat 6 uses an internal* `FormAuthenticatorValve` *to provide form-based authentication on security-protected resources; it also uses a* `StandardContextValve` *to invoke Context listeners. However, because most of these valves are internal to Tomcat 6 and are not generally accessible to administrators, they will not be covered here. Our focus in this section is only on Valves that are configurable by administrators.*

Access Log Implementation

Logging access to resources is a very common activity for Web server administrators. This can include either static resources (such as Web pages and graphic files) or dynamic resources (such as CGI, JSP, and servlets). In Tomcat 6, access logs can be generated by inserting an *Access Log Valve*.

Note that this standard Valve is completely separate and different from built-in logging support for Tomcat 6. The built-in logging support provides a method for capturing information, warning, and error messages for the Tomcat server itself. The standard Access Log Valve uses its own logic to examine each incoming request for Web resources (that is, the URL requested), and captures only the access request information to its own log file.

The typical format for the standard Access Log Valve is a well-known `common` log file format (for more information, see the following W3C page: `w3.org/Daemon/User/Config/Logging.html#common-logfile-format`).

This format is supported by almost all popular Web servers, including Apache. Analysis tools are widely available for the analysis of log files in the `common` log file format.

Scope of Log Files

The scope of logging depends on where the Access Log Valve is inserted. For example, to log all the access within a specific Web application, the Valve (the `<Valve>` element) should be placed in the `<Context>` container in the `$CATALINA_HOME/conf/server.xml` file. To log all the resource access within a virtual host across all Web applications, the Valve should be placed in the `<Host>` container. Finally, if you want to track all access to all resources on a particular instance of the Catalina engine (across all the virtual hosts in that Engine and across all the Web applications), the Valve should be placed in the `<Engine>` scope.

If you are using the default Access Log Valve (that is, if the `className` attribute is set to `org.apache.catalina.valves.AccessLogValve`), then you can specify the attributes shown in the following table.

Attribute	Description	Required?
`className`	The Java programming language executable class representing the Valve `org.apache.catalina.valves.AccessLogValve.`	Yes
`condition`	Provides conditional logging capability. The default is to log every request. If set to a value, the Valve will look for a request attribute with the specified name and log an entry only if that attribute exists. By adding application-level filters to the request chain, and having these filters set specific request attributes, application developers can control access logging conditionally.	No
`directory`	The directory in which the log files will be placed. Usually relative to the `$CATALINA_HOME`, but can also specify an absolute path instead. The default value is `logs`.	No
`fileDateFormat`	Use this in conjunction with a rotatable attribute to control the period of time between log rotation — and the name used for log files. To rotate the log at midnight every day, use yyyy-MM-dd — this is the default value for the attribute. To rotate the log every hour on the hour, use yyyy-MM-dd.HH. If the rotatable attribute is set to `false`, the value of this attribute is not used.	No
`pattern`	This attribute specifies the format used in the log. You can customize the format, or you can use the `common` format or the combined format (common log file entry plus referrer and user-agent logged). The default format is `common`. To customize the format, you can use any of the following pattern identifiers interspersed with a literal string in this pattern attribute: `%a` — Insert remote IP address. `%A` — Insert local IP address (of URL resource). `%b` — Insert bytes sent count, excluding HTTP headers; will show - if zero. `%B` — Insert bytes sent count, excluding HTTP headers.	No

Attribute	Description	Required?
	%h — Insert remote host name (or IP address if the resolveHosts attribute is set to false). %H — Insert the request protocol (HTTP). %l — Insert remote logical username (always -). %m — Insert request method such as GET or POST. %p — Insert the local TCP port on which the request is received. %q — Insert the query string of the request. %r — Insert the first line of the request. %s — Insert the HTTP status code of the response. %S — Insert the user session ID. %t — Insert the date and time in common log file format. %u — Insert the remote user that has been authenticated (otherwise, it is -). %U — Insert the URL path of the request. %v — Insert the name of the local virtual host from the request.	
resolveHosts	Determines if the log will contain host names via a reverse DNS lookup. This can take significant time if enabled. The default is disabled (false).	No
prefix	The prefix added to the name of the log file.	No
suffix	The suffix (extension) added to the name of the log file.	No
rotatable	A Boolean value. The default is true. True indicates support for rolling logs. A new log file will be created for each specified period of time. The period of time between log rotation is controlled by the fileDateFormat attribute. There should be very little reason for setting this attribute to false in production, as it may result in unbounded log file size.	No

Before you try out the following modification to the server.xml *file, and any time you modify* server.xml*, it is a good practice to first make a backup of the file. Just in case some configuration goes wrong and Tomcat 6 does not start, you can quickly restore a known-to-be-good configuration. A good way to name the file is to suffix it with the current date and time, such as* server.xml.200810011211.

Here is a practical example for the configuration of access logs. If you examine the default $CATALINA_HOME/conf/server.xml file, you will see a commented <Valve> entry that specifies the access log. It is placed immediately within the <Host name='localhost' ...> entry:

```
<!--
    <Valve className='org.apache.catalina.valves.FastCommonAccessLogValve'
           directory='logs'
           prefix='localhost_access_log.'
           suffix='.txt'
           pattern='common'
           resolveHosts='false'
    />
-->
```

> **If you do not see the commented section detailed here, type in the `<Valve>` entry manually, as shown next. It is possible that you may have modified some Tomcat settings using the `admin` application. Some versions of this administration tool will strip comments when changes are saved.**

Uncomment this entry and modify the `directory` and `prefix` attributes, as shown here:

```
      <Valve className='org.apache.catalina.valves.AccessLogValve'
             directory='wroxlogs'
             prefix='wroxtest_localhost_access_log.'
             suffix='.txt'
             pattern='common'
             resolveHosts='false'
      />
```

Now, create a `wroxlogs/` directory under the `$CATALINA_HOME` directory (the standard Valve will actually create it for you if you forget). Start/restart Tomcat 6, and then open a browser and point it to the following URL:

```
http://localhost:8080/
```

The default Tomcat 6 welcome page should be displayed. Shut down the Tomcat server and examine the `$CATALINA_HOME/wroxlogs/` directory. You will find the access logs created by the Valve.

Note that the log file's prefix is the same as the one configured in the `<Valve>` element. If you look inside this log file, you will see the access log entries to the home page and the associated GIF files, all in the `common` log file format:

```
127.0.0.1 - - [07/Aug/2007:12:15:12 -0500] "GET / HTTP/1.1" 200 7406
127.0.0.1 - - [07/Aug/2007:12:15:12 -0500] "GET /tomcat.gif HTTP/1.1" 200 1934
127.0.0.1 - - [07/Aug/2007:12:15:12 -0500] "GET /tomcat-power.gif HTTP/1.1" 200 2324
127.0.0.1 - - [07/Aug/2007:12:15:12 -0500] "GET /asf-logo-wide.gif HTTP/1.1" 200 5866
127.0.0.1 - - [07/Aug/2007:12:15:12 -0500] "GET /favicon.ico HTTP/1.1" 200 21630
```

You may want to experiment further with other attributes of the standard Access Log Valve. This can be done by modifying the `<Valve>` entry. If you are running multiple virtual hosts, you may want to try configuring the `<Valve>` at the `<Host>` level as well as the `<Context>` level to experiment with the scope difference. Logging involves disk writes, and will inherently introduce additional overhead into the server hosting Web applications.

This is particularly true for global Engine-wide or virtual host-wide request logging. For a production site, it is important to discuss and formulate an optimal logging strategy between developers and administrators, based on the application and demand of a server.

Single Sign-On Implementation

Another standard Valve that is frequently used is the *Single Sign-on Valve*. During conventional Web access, whenever a user of a Web application reaches a protected page, the user will be required to sign on. This is required for each Web application that may be accessed. Using single sign-on, it is possible to

eliminate this annoying repetition (provided the username and password are identical for each sign-on, and usually authenticating against the same Tomcat Realm).

The Single Sign-on Valve caches credentials (passwords) on the server side, and will invisibly authenticate users as they traverse between Web applications on a given virtual host. Without activating this Valve, the user will be prompted to authenticate for each and every protected Web application, even in cases where all applications use the same username and password. The credential is cached against a host-wide client session on the server side. This means that a single sign-on will be effective throughout the session.

Multiple Sign-On Without the Single Sign-On Valve

Before configuring the Single Sign-on Valve, you should understand what the user must go through without single sign-on. To do this, we must protect (enable authentication on) two Web applications within the same virtual host. We will do this for two of the default applications included with the Tomcat 6 distribution. These sample Web applications are as follows:

- The servlet and JSP examples Web application
- The Tomcat documentation Web application

Again, it is always a good practice to copy the original XML file to a backup before making a change.

To begin, edit the `web.xml` file in the `$CATALINA_HOME/webapps/tomcat-docs/WEB-INF` directory by adding the lines shown in the following listing:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <display-name>Tomcat Documentation</display-name>
  <description>
     Tomcat Documentation.
  </description>
    <security-constraint>
      <display-name>Example Security Constraint</display-name>
      <web-resource-collection>
         <web-resource-name>Protected Area</web-resource-name>
         <url-pattern>/*</url-pattern>
      </web-resource-collection>
      <auth-constraint>
         <role-name>tomcat</role-name>
      </auth-constraint>
    </security-constraint>
<login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>Single Sign-on Example</realm-name>
</login-config>
<security-role>
      <role-name>tomcat</role-name>
</security-role>
</web-app>
```

This modification protects all the pages of the documentation, via the <url-pattern>/*</url-pattern> element, by requiring an authentication for access. Only users belonging to the tomcat role can access these pages, as specified by the <auth-constraint> element. The <security-constraint> and <login-config> elements are part of the Servlet 2.4 (since 2.2) specification. Note that the <login-config> in this case specifies a BASIC authentication. This means the browser's Security dialog box will be used to obtain authentication information from the user.

Next, the Servlet and JSP examples Web application is protected. In the $CATALINA_HOME/webapps/examples/WEB-INF/web.xml file, make the following modifications:

```
<security-constraint>
      <display-name>Example Security Constraint</display-name>
      <web-resource-collection>
         <web-resource-name>Protected Area</web-resource-name>
         <!-- Define the context-relative URL(s) to be protected -->
         <url-pattern>/*</url-pattern>
         <!-- If you list http methods, only those methods are protected -->
         <http-method>DELETE</http-method>
         <http-method>GET</http-method>
         <http-method>POST</http-method>
         <http-method>PUT</http-method>
      </web-resource-collection>
      <auth-constraint>
         <!-- Anyone with one of the listed roles may access this area -->
         <role-name>tomcat</role-name>
         <role-name>role1</role-name>
      </auth-constraint>
</security-constraint>
<!-- Default login configuration uses form-based authentication -->
<login-config>
   <auth-method>FORM</auth-method>
   <realm-name>Example Form-Based Authentication Area</realm-name>
   <form-login-config>
      <form-login-page>/jsp/security/protected/login.jsp</form-login-page>
      <form-error-page>/jsp/security/protected/error.jsp</form-error-page>
   </form-login-config>
</login-config>
```

Here, the <url-pattern> element is modified to protect all the resources within the servlet and JSP examples Web application. Note that the <login-config> in this case specifies FORM-based authentication. This means a custom-created form will be used to obtain authentication information from the user, instead of the browser's Security dialog box.

Start Tomcat 6, and try to access the docs Web application via the following URL:

```
http://localhost:8080/docs/
```

Because BASIC authentication has been configured for docs, the browser prompts you to enter a username and password, as shown in Figure 6-2.

You can use tomcat for User Name, and tomcat for Password (the password is case-sensitive). This corresponds to one of the password entries added earlier in $CATALINA_HOME/conf/tomcat-users.xml, which is the default location of the XML file for loading the Memory Realm or UserDatabase Realm (see Chapter 14 for detailed coverage of security Realms). Once you authenticate successfully, you will

Figure 6-2: BASIC authentication for tomcat.docs application access

be able to reach the Tomcat documentation home page. Now, let's switch to another Web application on the same virtual host. Try the following URL to access the JSP examples:

```
http://localhost:8080/examples/jsp/
```

Note that you are requested to authenticate again, this time using a custom form that has been created as part of the Web application, as shown in Figure 6-3.

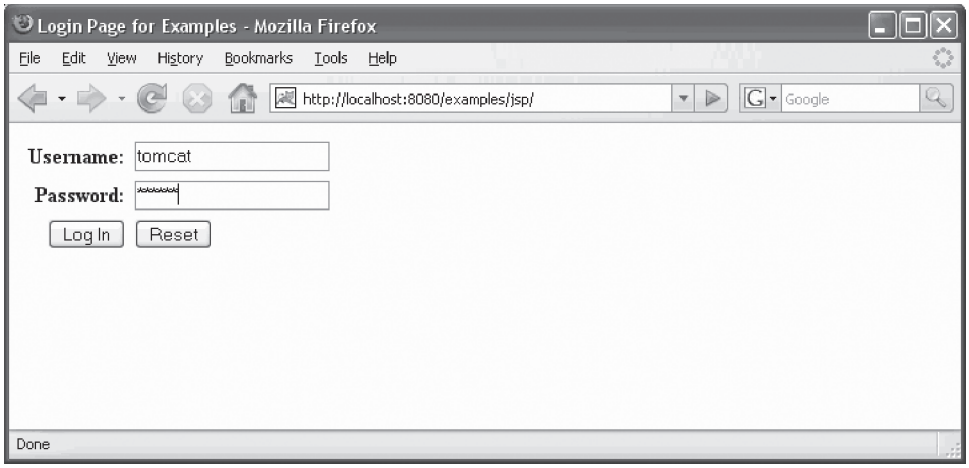

Figure 6-3: Authentication for accessing other applications on the same virtual host

If you enter `tomcat` for Username, and `tomcat` for Password again, you access the `examples` pages. In fact, if you have more Web applications that require authentication, the user will be prompted again when first accessing them unless you enable the Single Sign-on Valve.

Configuring a Single Sign-On Valve

To enable the Single Sign-on Valve, place a `<Valve>` element inside the `<Host>` element in the `$CATALINA_HOME/conf/server.xml` file. To specify the Single Sign-on Valve, set the `className`

attribute to the value `org.apache.catalina.authenticator.SingleSignOn`:

```
<Valve className='org.apache.catalina.authenticator.SingleSignOn'/>
```

Only one optional attribute is available with this Valve, `requireReauthentication`. This attribute defaults to `false`. When `requireReauthentication` is set to `true`, the Single Sign-on Valve performs actual re-authentication against each security Realm before accessing a protected resource. When this is set to `false`, by default, a session cookie is checked for authentication state and the Realm will not be contacted if the user is already signed on.

Restart Tomcat as well as your browser. (This is necessary because most browsers cache credentials for `BASIC` authentication.) Try accessing the two URLs again, in any order. This time, because the Single Sign-on Valve caches the access credentials across multiple Web applications on the same virtual host, you will be asked to enter the username and password only once. You can test this again by trying the URLs in a different order after restarting the browser (to clear the browser's password cache and create a new session).

> **Note that `BASIC` authentication was purposely not used for both applications because the client browser typically caches login usernames and passwords. This Valve is not as useful whenever all the applications use `BASIC` authentication (because the browser may already cache credentials for `BASIC` authentication, providing single sign-on capability in this special case). Therefore, depending on the authentication method used by Web applications, your mileage on the Single Sign-on Valve may vary. The Single Sign-on Valve is most effective when multiple authentication schemes are involved (common in most production scenarios).**

Form Authenticator Valve

In Figure 6-3, a custom form is used to authenticate the user. Inside Tomcat, the application Context has actually inserted a special Valve to handle this. This automatically inserted Valve is called the Form Authenticator Valve, and it presents the form and handles form submission for user authentication.

In most cases, you will not need to modify the default configuration of this Valve. However, in special cases where you need to accept the username and password in a text encoding that is different from the request's text encoding, you can override the default:

```
<Valve className="org.apache.catalina.authenticator.FormAuthenticator"
characterEncoding="UTF-8"/>
```

Restricting Access via a Request Filter

A *Request filter* is a very useful Valve that enables you to block or filter specific client requests. This Valve is useful for implementing policies that are based on the characteristics of requests passing through it. These filters are discussed next.

Remote Address Filter

If the `className` attribute of the `<Valve>` component has the value `org.apache.catalina.valves.RemoteAddrValve`, then a *Remote Address Filter* is created. A Remote Address Filter enables the

administrator to specify a list of IP addresses (or regular expressions representing IP addresses) from which to accept or deny requests. Any denied request is not passed through the Valve, effectively blocking it completely from further processing. The following table describes the attributes allowed with the Remote Address Filter.

Attribute	Description	Required?
`className`	The Java programming language executable class representing the Valve — typically, `org.apache.catalina.valves.RemoteAddrValve`	Yes
`allow`	An IP address specified using a regular expression that matches the address of incoming requests.	No
`deny`	An IP address specified using a regular expression that matches the address.	No

This Valve examines the IP address of the client's request against its `allow/deny` list, and attempts to match the specified regular expression representing IP addresses. Any address that does match the `allow` attribute will be passed through to downstream components. If `allow` is not specified, all IP addresses other than the ones specified in `deny` are allowed.

Remote Host Filter

If the `className` attribute of the `<Valve>` component has the value `org.apache.catalina.valves.RemoteHostValve`, a *Remote Host Filter* is created. A Remote Host Filter functions like the Remote Address Filter, except the filtering performed is based on host names, rather than IP addresses. The allowed attributes are `allow` and `deny`, but the regular expression specified is used to match a host name, rather than an IP address.

> **Use of the Remote Host Filter requires a reverse DNS lookup. Therefore, the DNS service must be accessible from the server side. In addition, you must be careful to specify all variants (or use a regular expression) of host names that a particular remote host can assume. For example, if a host has only two names, `printserver.wrox.com` and `charlie.wrox.com`, you should be careful to use `printserver.wrox.com,charlie.wrox.com` to match it; `*.wrox.com` will also work, but will potentially match many other hosts.**

Configuring Request Filter Valves

Let's look at the details of configuring both Request Filter Valves discussed in the preceding section. Before starting Tomcat, add the following line to the `$CATALINA_HOME/conf/server.xml` file inside the '`localhost`' `<Host>` container, and then start Tomcat:

```
<Valve className='org.apache.catalina.valves.RemoteAddrValve'
       allow='121.121.121.*,111.111.111.*'/>
```

This will set up a Request Filter Valve to allow only requests from the two subnets: `121.121.121.*` and `111.111.111.*`.

Now, try accessing the following URL:

```
http://localhost:8080/examples/jsp/
```

The list of allowed IP addresses does not have an entry that matches the IP (`127.0.0.1`); therefore, the request is filtered out. The server returns an HTTP "Forbidden" 403 error and you get a blank page.

If you need to have a custom page returned when access is denied, you must use Servlet 2.4 filters within a Web application instead. Custom error pages are configurable inside the deployment descriptor.

Now, edit the previous line again to include your IP address:

```
<Valve className='org.apache.catalina.valves.RemoteAddrValve'
       allow='121.121.121.*,127.*.*.*'/>
```

Restart Tomcat, and now the URL can be accessed again, as the IP is explicitly enabled by the `allow` list.

You can also explicitly deny access by changing the line, as shown here:

```
<Valve className='org.apache.catalina.valves.RemoteAddrValve'
       deny='127.0.0.1'/>
```

When you try to access the URL again, a blank page is returned, as the request is filtered out again.

The Remote Host Filter works identically, but with host names instead. You can try it out by simply editing the previous configuration line as follows:

```
<Valve className='org.apache.catalina.valves.RemoteHostValve'
       allow='*.wrox.com'/>
```

Notice the change in `className` from `org.apache.catalina.valves.RemoteAddrValve` to `org.apache.Catalina.valves.RemoteHostValve`, and that the `deny` list now contains a host name instead of an IP address. Restart Tomcat and try accessing the URL again.

The access fails and you get a blank page, as only hosts from `wrox.com` with names that are DNS-resolvable are explicitly allowed.

The Request Filter Valve can be quite effective in implementing a security policy, although if filtering on a physical server level is desired, router-based hardware filtering may be more suitable. Regardless, the Request Filter Valve is handy for temporarily removing access to specific remote client(s).

Request Dumper Valve

A lesser-known standard (useful for debugging Web applications that most administrators/users typically overlook) is called the *Request Dumper Valve*. This Valve dumps the headers and cookies of requests and responses to the log of the associated component.

For administrators, it serves the following two purposes:

- It visually illustrates how the scope of a Valve affects the requests that are processed.

- It is used to debug the actions of other Valves (that is, when a Request Filter Valve does not appear to work) or to request processing components.

To configure a Request Dumper Valve, simply add the following to the `<Context>`, `<Host>`, or `<Engine>` element:

```
<Valve className='org.apache.catalina.valves.RequestDumperValve'/>
```

Make sure you remove any definitions of `RemoteAddrValve` or `RemoteHostValve` before using this Valve. Otherwise, you may not gain access to the application at all.

Persistent Sessions

Tomcat 6 features a *Persistent Session Manager* to manage the backup of user sessions onto disk. This manager is not configured by default. This section covers the need for the Persistent Session Manager, as well as its configuration details.

The Need for Persistent Sessions

When Tomcat 6 is shut down, typically all session information is lost. Furthermore, sessions that are idle consume valuable working memory until session timeout, which can be a long period, as some users may leave their computers in the middle of a session.

With Persistent Session Manager, the following features can be enabled:

- Sessions that are inactive can be configured to be *swapped onto disk*, thereby releasing the memory consumed by them, and making memory available for other active sessions.
- When Tomcat is shut down, all the current sessions are *saved to disk*. Upon restart, the saved sessions are restored.
- Sessions lasting beyond a specified threshold period are *automatically backed up* on disk, enabling the system to survive an unexpected crash.

The last feature listed here enables continuous reliable execution of the Web application despite minor server failure (crash), and goes a long way toward enhancing the availability and robustness of the system. However, the Persistent Session Manager is still considered to be of experimental quality, rather than production quality, as of this writing.

Configuring a Persistent Session Manager

The Persistent Session Manager is configured through the `<Manager>` element in the Context descriptor of a Web application. This element must be configured as a subelement of the `<Context>` element within the application's Context descriptor XML file. The `<Context>` element can placed in the global `context.xml` found in the `$CATALINA_HOME/conf` directory (applied globally to all Web applications in the server) or in a `context.xml` file placed in the `META-INF` directory of the application's WAR file.

The <Manager> Element

The following table describes the most common attributes of the `<Manager>` element that are available for configuration.

Attribute	Description	Required?
`className`	The Java programming language class that implements the Persistent Session Manager. This should be `org.apache.catalina.session.PersistentManager`.	Yes
`algorithm`	The algorithm used for generating the session IDs. Default of `MD5` should be adequate for almost all cases.	No
`distributable`	Default is `false`. A `<distributable/>` element in an application's `web.xml` is equivalent to setting this to `true`. When `true`, the Session Manager will enforce that all session attributes be serializable.	No
`entropy`	The seed (string) value used in generating randomized session IDs by the persistence manager. Default is a random computed value. You can specify any long string as a seed.	No
`maxActiveSessions`	The ceiling on the number of active sessions before swapping out of the session via the Persistent Session Manager begins. The default value of –1 allows an unlimited number of active sessions.	No
`minIdleSwap`	The minimum number of seconds before a session will be considered for swapping. The default value of –1 enables swapping at any time. Make sure this is `maxIdleSwap` if you specify it.	No
`maxIdleSwap`	The maximum number of seconds before a session is eligible to be swapped out to the Store. Used with `minIdleSwap` to tune the session persistence mechanism. The default value is –1, and the session will be swapped without an eligibility check.	No
`maxIdleBackup`	The number of seconds since a session was last active before it is backed up on the Store. Note that sessions that are backed up are *not* removed from memory — they are not swapped. This can be used to avert a sudden crash, as the backed up sessions will be restored from the Store upon the next startup. The default value of –1 will disable the backup action altogether.	No
`randomClass`	The implementation of `java.util.Random` to use. By default, `java.security.SecureRandom` is used.	No
`saveOnRestart`	If this is set to `true`, Tomcat will save all the active sessions to the Store upon shutdown, and will reload the sessions (except the expired ones) from the Store on startup. The default is `true`.	No
`sessionIdLength`	The length of the session ID created by the Session Manager instance. Default is 16.	No

The <Manager> element can have only one subelement, as described in the following table.

Subelement	Description	How Many?
Store	Used by the Persistent Session Manager to determine how and where to save the session. Currently, the only options available for a Store implementation are org.apache.catalina.session.FileStore or org.apache.catalina.session.JDBCStore.	1

Store uses object serialization to store the session. The following hands-on example uses the FileStore Store implementation. By default, the FileStore's serialized session information is placed under the $CATALINA_HOME/work/<*service name*>/<*host name*>/<*web-app name*>/ directory. You can customize the directory for use by setting the directory attribute of the Store element.

Hands-On Configuration with the Persistent Session Manager

To configure the Persistent Session Manager, it is necessary to add a <Manager> element definition into the Context descriptor of the Web application. In this example, the default servlet and JSP examples Web application will be used.

The easiest way to add a Context descriptor to the examples application is to create a $CATALINA_HOME/webapps/examples/META-INF/context.xml file containing the following:

```
<Context>
   <Manager className='org.apache.catalina.session.PersistentManager'
         saveOnRestart='true'
         maxActiveSessions='3'
         minIdleSwap='0'
         maxIdleSwap='60'
         maxIdleBackup='0'>
   <Store className='org.apache.catalina.session.FileStore'/>
   </Manager>
</Context>
```

This configures a Persistent Session Manager that will allow up to three active sessions before activating session swapping. Any session is available for swapping at any time. All idle sessions will be swapped within 60 seconds. An active session is backed up regularly; a value of 0 indicates that sessions should be backed up immediately after being used.

Start Tomcat 6, start a session, and view the session information by going to the following URL:

```
http://localhost:8080/examples/servlets/servlet/SessionExample
```

Note on a piece of paper the session ID and the start date of your session.

Now, wait for about two minutes for the Persistent Session Manager to go to work. At this point, simulate a crash via an ungraceful shutdown. This can be done by a Ctrl+C in the Tomcat window. Make sure you do not close the browser.

Next, start Tomcat again. When Tomcat starts, it will restore all the sessions that were backed up. Try the following URL again:

```
http://localhost:8080/examples/servlets/servlet/SessionExample
```

Note that the session ID and backed-up session information are identical to what appeared before the (simulated) crash. In effect, our Tomcat server has survived an unexpected sudden crash. The Persistent Session Manager already backed up the session by the time you crashed Tomcat. Therefore, when you restart Tomcat, it restores the session from the backed-up store, and you resume the previous session.

To see where the persisted sessions are stored, go to the `$CATALINA_HOME/work/Catalina/localhost/examples/` directory and look for file names with the `.session` extension.

JNDI Resource Configuration

Within either the `server.xml` configuration file or the `context.xml` file, *Java Naming and Directory Interface (JNDI)* resources can be defined; and they may be accessed in a standard J2EE-compliant manner by any Web applications. This section provides a brief introduction to JNDI. Examples illustrate how it is used and the type of administrative requests that developers typically make. A basic understanding of these requests is important to any administrator because they have an impact on how the JNDI resources must be configured.

What Is JNDI?

JNDI is an API used to look up information pertaining to the network (via naming and directory services). JNDI is designed to work with any compatible naming and directory service, regardless of its native interface API. Some common information that can be obtained through JNDI includes (but is not restricted to) the following:

- Username and password (authentication)
- Access control policy (who can access what)
- Organizational directories
- Servers (e-mail, database, and so on)
- Printers
- Other objects or resources

Before the advent of JNDI, developers had to program specifically to a particular network's directory service. On Microsoft-based networks, they programmed to NT Domains or the Active Directory Service Interface (ADSI). On Solaris/Unix networks, they programmed to the Network Information Service (NIS). On Novell networks, they programmed to the Netware Directory Service (NDS). This made the programming even more complex because each of the network directory services assumes a different naming convention for the resources/information that it stores, and has completely different programming interfaces (APIs) to search for and locate this information.

Figure 6-4 shows how JNDI unifies directory service access across different networks.

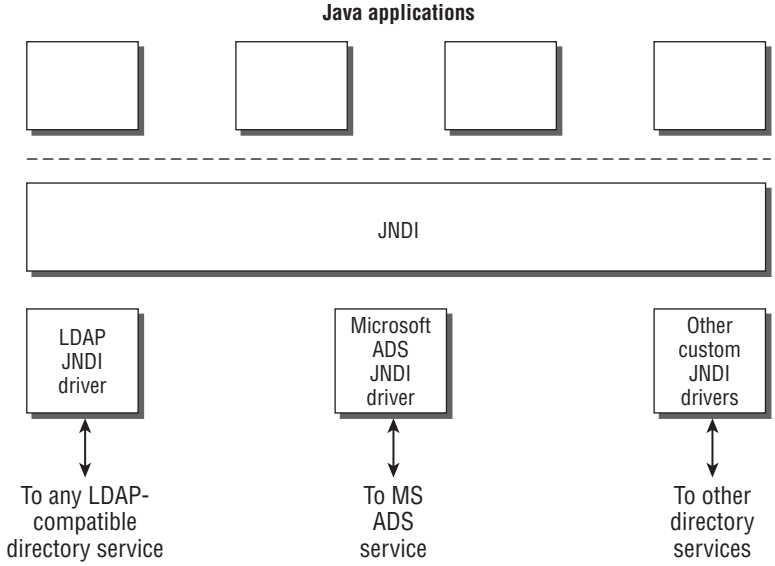

Figure 6-4: JNDI unifies access of different directory services

As shown in Figure 6-4, JNDI is the top layer that provides a uniform programming interface to applications, while translating the API commands to the network-specific operations that are sent out through its plug-in drivers. In fact, many of the modern directory services support the Lightweight Directory Access Protocol (LDAP). JNDI often gains compatibility with new or legacy directory services through its LDAP driver.

> *Beyond providing interfaces to existing directory services, JNDI has become a standard way for Java applications (especially in the context of J2EE) to locate network resources. That is, even if there is no physical directory service involved over the network, many of the standard Java APIs have adopted JNDI as the de facto way of obtaining network resources. Developers expect to use JNDI to obtain these Tomcat-managed resources. The next section discusses two such examples.*

Tomcat and JNDI

The role of the Tomcat server, with respect to supporting JNDI, is quite interesting. In fact, the role of Tomcat in this case is only to provide the JNDI lookup facilities (acting as a sort of go-between to provide a standard interface) for any Web application. Tomcat is a J2EE-compliant and Servlet 2.4–compliant server that will facilitate the use of JNDI by hosted Web applications, as shown in Figure 6-5.

In Figure 6-5, the Web application running inside Tomcat is retrieving certain JNDI resources through standard programming convention and APIs (specified by JNDI and Servlet 2.4 specifications). This enables the requests to be placed in a manner independent of the application server — enabling the Web applications to be portable across different vendors' application servers.

The Tomcat container intercepts the standard JNDI requests from the application. To fulfill these JNDI API requests, Tomcat must check its set of preconfigured resources (in the `server.xml` or `context.xml` file) to determine what needs to be passed back to the application. Tomcat essentially provides

JNDI emulation service for accessing these resources. Tomcat's administrator must configure these resources.

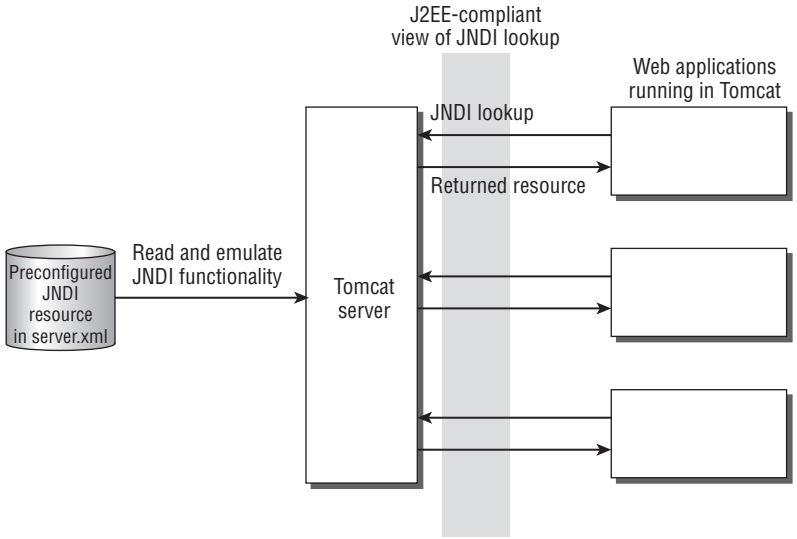

Figure 6-5: Tomcat facilitates resource acquisition by emulating JNDI

Typical Tomcat JNDI Resources

Two interesting resources that are accessed via JNDI requests from Web applications include the following:

- A JDBC DataSource
- A JavaMail session

JDBC is a well-known standard API that enables application programmers to access relational databases (such as MySQL, Oracle, and SQL Server) in a uniform and standard way. To access data from these relational databases, the application must first obtain an initial Context `DataSource` object. (See Chapter 13 for extensive JDBC coverage. For now, it is sufficient to view a `DataSource` as a class from which it is possible to obtain connections for a remote database.)

JavaMail is another well-known standard API that provides an interface to access e-mail client capabilities (that is, to create and send mail) across different methods of handling e-mail, in a uniform and standard manner. For a Web application to access mail servers and send mail, the application must first obtain a JavaMail `Session` object.

The JDBC 3 and JavaMail 1.3 specifications are synchronized with the J2EE 1.4 specification. In both cases, it is the responsibility of the container (application server) to provide Web applications with JNDI access to these resources. The administrator needs to configure these resources for Tomcat 6 to find them, passing them through to the requesting Web application by emulating JNDI action.

Configuring Resources via JNDI

For JNDI, you have the following three options for configuring the resource within the hierarchy of Tomcat configuration components:

- At the server's global `<GlobalNamingResources>` level
- At the virtual host's `<DefaultContext>` level
- At the `<Context>` level associated with a single Web application, typically residing in the application's Context descriptor XML file (`context.xml`)

Any JNDI resource configured at the `<DefaultContext>` level will be available to all Web applications running on the same virtual host, whereas any JNDI resource configured at the `<Context>` level will be available only within the specific Web application associated with that Context.

Resources configured at the `<GlobalNamingResources>` level are available server-wide (across all services and engines). These resources can then be referred to in subsequent resource configurations via `<ResourceLink>` elements.

You can add the subelements described in the following table inside the `<Context>` or `<DefaultContext>` element to support and configure JNDI resources.

Subelement Name	Description	How Many?
`Environment`	Creates environment entries available from the JNDI `InitialContext` that Tomcat will supply to an application	0 or more
`Resource`	Provides the name of a datatype of a JNDI resource to the application	0 or more
`ResourceParams`	Specifies the Java programming language class that is used to create the resources, and specifies a configuration JavaBean	0 or more
`ResourceLink`	Adds a link to a resource defined in the `<GlobalNamingResource>` element, which is server-wide	0 or more

It is also possible for developers to directly embed environment or resource parameters into their Web applications. This is done by defining `<env-entry>`, `<resource-env-entry>`, and `<ResourceParams>` elements inside the `web.xml` descriptor. This will make a resource specific to a Web application. However, the `web.xml` deployment descriptor (typically part of the WAR archive) must be changed each time a change occurs in the resource information.

The following sections examine each of the JNDI supporting subelements.

The <Environment> Element

The `<Environment>` element is used to pass named data values (such as environment variables in a command shell) to the Web applications. Web applications can access these values through the JNDI Context. The following table describes the available attributes for an `<Environment>` element.

Attribute	Description	Required?
`name`	The JNDI name for this element.	Yes
`description`	Text description for this element.	No
`override`	Application programmers can use the `<env-entry>` element to override the one defined here. You can disable the override by setting it to `false`.	No
`type`	Java class name of the datatype represented by this element.	Yes
`value`	The actual value of the environment entry.	Yes

For example, the following will add a JNDI entry named `maxUsers` with a value of `100`:

```
<Environment name='maxUsers' type='java.lang.Integer' value='100' />
```

The <Resource> Element

The `<Resource>` element is used to pass a reference via resource managers (classes that manage and assign resources — such as JDBC connections) to Web applications using a name in simple text. A Web application can access the reference to the resource manager through a lookup based on the textual name using the JNDI context. The following table describes the attributes a `<Resource>` element can have.

Attribute	Description	Required?
`auth`	Indicates who does the authentication. If the value is `application`, then the application itself must sign on with the resource manager. If the value is `container`, then the container does a sign-on with the resource manager.	No
`description`	Text description for this element.	No
`name`	Name of the resource.	Yes
`scope`	Can be either `Shareable` or `Unshareable`; determines if the resource can be shared.	No
`type`	Java class name of the datatype represented by this resource.	Yes

For example, the following will add a `UserDatabase` implementation (for storing authentication and role information on Tomcat 6):

```
<Resource name='myDatabase'
          type='org.apache.catalina.UserDatabase'>
</Resource>
```

The <ResourceParams> Element

The <ResourceParams> element associates parameters with the resource manager already configured in a <Resource> element. This element is often used to configure the resource manager. For example, if the <Resource> is a JDBC DataSource, the <ResourceParams> may contain the Relational Database Management Server (RDBMS) server location, login name, and password to use. The <ResourceParams> element can contain the attribute shown in the following table.

Attribute	Description	Required?
name	Name of corresponding resource	Yes

Each <ResourceParams> element can contain one or more <name>/<value> subelements, expressed as follows:

```
<ResourceParams name='jdbc/wroxDatabase'>
   <parameter>
      <name>password</name>
      <value>wrox123</value>
   </parameter>
</ResourceParams>
```

The <ResourceLink> Element

The <ResourceLink> element refers to a previously configured JNDI resource (typically in the <GlobalNamingResource> subelement associated with a server), making these resources available to all <Service>, <Host>, and <Context> components. This enables resources to be defined and shared across servers or globally. A <ResourceLink> element can have the attributes described in the following table.

Attribute	Description	Required?
global	The name of the resource being linked to	Yes
name	The name of the resource, accessible by Web application via JNDI lookup	Yes
type	The Java programming language class name indicating the type of resource returned	Yes

For example, if the UserDatabase <Resource> element is already defined in the server's <GlobalNamingResource> subelement (see the <Resource> element example earlier), then it can be referred to within a <Context> element of a Web application using the following:

```
<ResourceLink name='localDatabase' global='myDatabase'
              type='org.apache.catalina.UserDatabase'/>
```

This entry will link the previously defined UserDatabase instance (named myDatabase in <GlobalNamingResource>) to the JNDI addressable resource called localDatabase. The Web application can perform a JNDI lookup for localDatabase and obtain access to the UserDatabase instance.

The next section shows how to apply these elements to configure a JDBC DataSource and JavaMail session.

Configuring a JDBC DataSource

JDBC features, including connections pooling, are directly supported by Tomcat 6. JDBC is discussed at length in Chapter 13. For now, it is necessary to know only how JDBC connections (as a JNDI resource) can be passed to Web applications.

Your JDBC driver can be placed in the `$CATALINA_HOME/lib/` directory. This enables the Tomcat server and your applications to find and access this driver.

Finally, you must configure the JNDI resource factory using `<Resource>` and `<ResourceParams>` elements. In this case, you are configuring the database factory to use the MySQL database, with an application-wide scope. This instance of the DataSource will be available only to the associated application. The definition needs to be placed into a `context.xml` file under the `webapps/[application name]/META-INF` directory:

```
<Context>
  ...
  <Resource name='jdbc/wroxTC6' auth='Container'
            type='javax.sql.DataSource'/>
```

This segment configures Tomcat's built-in JDBC DataSource factory. The built-in DataSource factory implementation in Tomcat is `org.apache.tomcat.dbcp.dbcp.BasicDataSourceFactory`. A DataSource factory is a class from which new instances of DataSource objects can be obtained. With this factory, the configuration parameters described in the following table are possible.

Parameter	Description
`driverClassName`	Java programming language class name of the JDBC driver. This driver should be placed in the `$CATALINA_HOME/lib` directory for easy location by the DataSource factory code.
`maxActive`	The maximum number of active connections in this pool.
`maxIdle`	The maximum number of idle connections in this pool.
`maxWait`	In milliseconds, indicates the maximum wait for a connection by the DataSource factory before throwing an exception.
`user`	The user ID used to log on to the database.
`password`	The password used to log on to the database.
`url`	The JDBC-compatible URL specifies the database instance to be used.
`validationQuery`	An optional SQL query used to validate a connection. Essentially, the factory will perform this query to ensure that rows are returned before considering the connection valid.

For example, you can parameterize the defined JDBC resource by using the following `<ResourceParams>` elements:

```
<ResourceParams name='jdbc/WroxTC6'>
    <parameter>
        <name>driverClassName</name>
        <value> com.mysql.jdbc.Driver </value>
    </parameter>
    <parameter>
        <name>url</name>
        <value>jdbc:mysql://localhost/wroxtomcat</value>
    </parameter>
    <parameter>
        <name>username</name>
        <value>empro</value>
    </parameter>
    <parameter>
        <name>password</name>
        <value>empass</value>
    </parameter>
    <parameter>
        <name>maxActive</name>
        <value>20</value>
    </parameter>
    <parameter>
        <name>maxIdle</name>
        <value>30000</value>
    </parameter>
    <parameter>
        <name>maxWait</name>
        <value>100</value>
    </parameter>
</ResourceParams>
```

In addition to the preceding configuration, the developer must declare the use of the resource in a deployment descriptor (`web.xml`) using a `<resource-ref>` element, as shown in the following example:

```
<resource-ref>
  <res-ref-name> jdbc/WroxTC6</res-ref-name>
  <res-type> javax.sql.DataSource </res-type>
  <res-auth> Container </res-auth>
</resource-ref>
```

Within the Web applications, the DataSource can be looked up relative to the `java:comp/env` naming context. The code used will be similar to the following:

```
private final Object lock = new Object();
...
Connection myConn = null;
synchronized(lock) {
  Context myInitialContext = new InitialContext();
  Context localContext = (Context) myInitialContext('java:comp/env');
  DataSource myDataSource = (DataSource)
localContext.lookup('jdbc/wroxTC6');
  myConn = myDataSource.getConnection();
}
...
```

Note the use of a lock around this lookup code. This ensures serialized access to the JNDI lookup code. There is production evidence on very highly loaded systems that the code may not be totally thread-safe. (Thanks to Anne Horton for this valuable information.)

At this point, `myConn` contains an instance of a database connection, which can be used to access the MySQL database immediately.

Configuring Mail Sessions

JavaMail is a standard programming API used by Java developers to create and send e-mail. Tomcat 6 supports JavaMail by providing the JNDI configuration of a JavaMail session as a resource, using its own "factory code" to create a JavaMail session for the Web application. This enables any Web applications running under Tomcat to use JNDI to look up and use the session.

The example in the following section shows how to send e-mail from within a JSP using a JavaMail session configured as a JNDI resource. The JSP will post a form to a collaborating servlet. The servlet will use the configured JavaMail session to send the actual e-mail. It takes advantage of the `examples` Web application that is distributed with Tomcat 6.

Adding a Resource Definition to the Application Context Descriptor

The first step is to configure a mail session as a JNDI resource. In the Context descriptor of the `examples` Web application (the `$CATALINA_HOME/webapps/examples/META-INF/context.xml` file), add the following resource definition inside the `<Context>` element. If this file and directory do not exist, add them explicitly:

```
<Context privileged='true'>
 <Resource name='mail/Session' auth='Container' type='javax.mail.Session'/>
 <ResourceParams name='mail/Session'>
            <parameter>
              <name>mail.smtp.host</name>
              <value>localhost</value>
            </parameter>
 </ResourceParams>
</Context>
```

This will configure the JNDI `mail/Session` Context, referring to an SMTP server running on `localhost`. If you are connecting to a remote SMTP server, change the value of `localhost` to the name or IP address of your server. You can also modify the port used (if it is not at the standard port 25) by setting the `mail.smtp.port` parameter.

Adding a Reference to a Mail Session Resource in the Deployment Descriptor

In the deployment descriptor (the `$CATALINA_HOME/webapps/jsp-examples/WEB-INF/web.xml` file), you must declare a reference to the JNDI resource. Add the following lines after the `<security-role>` declarations, but before the `<env-entry>` descriptions in the `web.xml` file:

```
...
</security-role>
<resource-ref>
      <res-ref-name>mail/Session</res-ref-name>
      <res-type>javax.mail.Session</res-type>
      <res-auth>Container</res-auth>
</resource-ref>
<env-entry>
...
```

Downloading and Installing JavaMail 1.3.3 and the JavaBeans Activation Framework 1.1 Libraries

Check the `%CATALINA_HOME%/lib` directory to determine whether you have `mail.jar` and `activation.jar` libraries there. These are the JavaMail and JavaMail-dependent JAF libraries. If they are not there, they will need to be downloaded.

JavaMail support is a part of the J2EE download, or it can be obtained as an optional download. You can find the required `mail.jar` library as part of the JavaMail distribution, downloadable from the following URL:

```
http://java.sun.com/products/javamail/downloads/index.html
```

The latest version available at the time of this writing is JavaMail 1.3.3. Because JavaMail 1.3.3 depends on JAF, you will also need to download the JavaBeans Activation Framework from the following URL:

```
http://java.sun.com/products/javabeans/jaf/index.jsp
```

The library that you will need from this download is `activation.jar`.

Compiling and Configuring the SendMailServlet

The code distribution includes the binaries for `SendMailServlet`. This sample servlet may already be part of your Tomcat 6 distribution. If not, copy the `SendMailServlet.class` file into the `webapps/examples/WEB-INF/classes` directory.

To configure the servlet, add the following servlet definition and mapping to the `web.xml` deployment descriptor of the `jsp-examples` Web application:

```
    <servlet>
         <servlet-name>SendMailServlet</servlet-name>
         <servlet-class>SendMailServlet</servlet-class>
    </servlet>
    <servlet-mapping>
         <servlet-name>SendMailServlet</servlet-name>
         <url-pattern>/mail/SendMailServlet</url-pattern>
    </servlet-mapping>
```

Creating the sendmail.jsp JSP

If `sendmail.jsp` is not included with your Tomcat 6 distribution, copy it from the code distribution to the `%CATALINA_HOME%/webapps/examples/jsp/mail` subdirectory (create this directory if necessary).

This is the JSP that will accept user input and submit the e-mail details to the `SendMailServlet` for sending.

Sending E-mail via JavaMail Sessions

Start Tomcat 6, and you can test the example that uses JavaMail to send e-mail. Use the following URL:

```
http://localhost:8080/examples/jsp/mail/sendmail.jsp
```

Figure 6-6 shows the JSP-generated form that you can fill out to send e-mail.

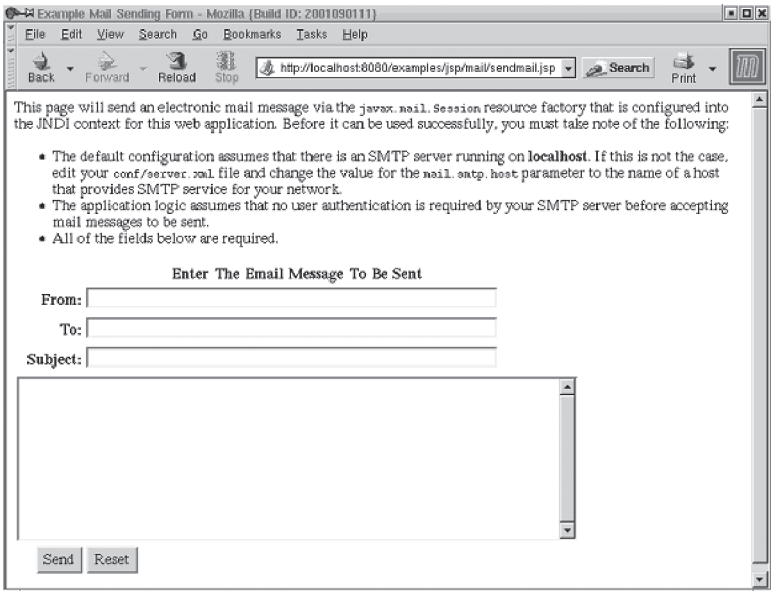
Example Mail Sending Form - Mozilla {Build ID: 2001090111}
File Edit View Search Go Bookmarks Tasks Help
Back Forward Reload Stop
http://localhost:8080/examples/jsp/mail/sendmail.jsp
Search
Print
This page will send an electronic mail message via the javax.mail.Session resource factory that is configured into the JNDI context for this web application. Before it can be used successfully, you must take note of the following:
The default configuration assumes that there is an SMTP server running on localhost. If this is not the case, edit your conf/server.xml file and change the value for the mail.smtp.host parameter to the name of a host that provides SMTP service for your network.
The application logic assumes that no user authentication is required by your SMTP server before accepting mail messages to be sent.
All of the fields below are required.
Enter The Email Message To Be Sent
From:
To:
Subject:
Send
Reset

Figure 6-6: Accessing JavaMail via JNDI

You can fill out the form shown in Figure 6-6 to actually send an e-mail message (assuming that you have the SMTP server configured properly).

This JSP collects information for an e-mail message from the user, and then submits it to the `SendMailServlet` for processing and sending. The following code shows how `SendMailServlet` (or other Web application code) can look up and utilize the JNDI mail session:

```
private final Object lock = new Object();
...
// Acquire our JavaMail session object
Session session = null;
Synchronized(lock) {
   Context initCtx = new InitialContext();
   Context envCtx = (Context) initCtx.lookup('java:comp/env');
   session=(Session) envCtx.lookup('mail/Session');
}
...
```

Configuring Lifecycle Listeners

Many top-level and nested components in the Tomcat 6 architecture (including Server, Service, Realm, and so on) support the configuration of *lifecycle listeners*. Lifecycle listeners are Java code modules that can be hooked into the server logic and executed during specific moments during the lifecycle of a component. This capability enables new custom functionality to be introduced to the Tomcat 6 server without having to change the core server code base.

With Tomcat 6, the only explicit use of a lifecycle listener is to insert code that enables the server to be managed remotely (through JMX support). Using a simple example, the following section shows how this support code is configured.

Lifecycle Events Sent by Tomcat Components

Lifecycle listeners are customized code that listens to specific lifecycle events. Lifecycle events are sent by a component, to any configured listener, at well-defined points in a component's lifecycle. These points include the following:

- Just before component startup
- During component startup
- Just after component startup
- Just before component stop
- During component stop
- Just after component stop

Developers may use lifecycle listeners to add new processing logic to the Tomcat server. As an administrator, you can add these custom listeners by creating a `<Listener>` XML element within the associated component.

The <Listener> Element

You can add a lifecycle listener to a Tomcat component (if the component supports lifecycle listeners) by configuring a `<Listener>` XML element within the XML definition of the component. Most Tomcat 6 architectural components support lifecycle listeners.

In Tomcat 6, listeners for the `<Server>` component are used to create JMX MBeans that represent runtime server structures and global resources. *JMX MBeans* are objects that enable Tomcat components, structures, and resources to be monitored or accessed via an external management system. Chapter 17 provides more extensive coverage of JMX.

More specifically, Tomcat 6 uses the following default `server.xml` fragment to add JMX MBean support:

```
<Server port='8005' shutdown='SHUTDOWN' debug='0'>
  <Listener className="org.apache.catalina.core.AprLifecycleListener" />
  <Listener className='org.apache.catalina.mbeans.ServerLifecycleListener'/>
  <Listener className='org.apache.catalina.mbeans.GlobalResourcesLifecycleListener'/>
<Listener className="org.apache.catalina.storeconfig.StoreConfigLifecycleListener"/>
```

Although the XML configuration syntax of a lifecycle listener is similar to the configuration of a nested component inside a container, technically, a listener is *not* an architectural component (and definitely not a nested component). A lifecycle listener should be thought of as an extended attribute of the containing XML element. Lifecycle listeners are configured as XML subelements instead of XML element attributes for the following reasons:

- Multiple lifecycle listeners can be associated with a single component.
- Each listener can be configured with its own set of attributes.

The `<Listener>` element, representing a lifecycle listener, can be configured with the attributes described in the following table.

Attribute	Description	Required?
`className`	The Java programming language class that implements the listener logic. This class must implement the `org.apache.catalina.LifecycleListener` Java interface.	Yes
`descriptors`	A semicolon-separated list of MBean descriptor XML files. This attribute is used to provide JMX compatibility (see Chapter 17 on Tomcat's JMX support) for custom components (that is, custom Valves and custom Realms).	No

Tomcat 6 Lifecycle Listeners Configuration

Tomcat 6 has two custom listener classes that will intercept lifecycle events and create (or destroy) management objects (called MBeans) to support external management of Tomcat. These two custom listener classes are as follows:

- `org.apache.catalina.mbeans.ServerLifecycle`: A listener to create/destroy MBeans for management of Tomcat architectural components.
- `org.apache.catalina.mbeans.GlobalResourcesLifecycle`: A listener to create/destroy MBeans for management of any global resources that may be externally manageable.

Displaying MBeans Created by Lifecycle Listeners Using the Manager JMX Proxy

The two lifecycle listeners are configured in the default `server.xml` file. You can see the result of the created MBeans by using the `manager` application's JMX proxy servlet. Try the following URL:

```
http://localhost:8080/manager/jmxproxy/?qry=*%3Atype%3DRole%2C*
```

This proxy servlet enables you to query specific MBeans that are created within the Tomcat 6 server. MBeans representing roles are created only by the lifecycle listeners described earlier. When these lifecycle listeners are not hooked in, the *roles'* MBeans are not created.

The preceding query will enumerate all the roles' MBeans. The query should return a list of the four manageable objects (MBeans) for the roles that are defined in `tomcat-users.xml`, as shown in Figure 6-7.

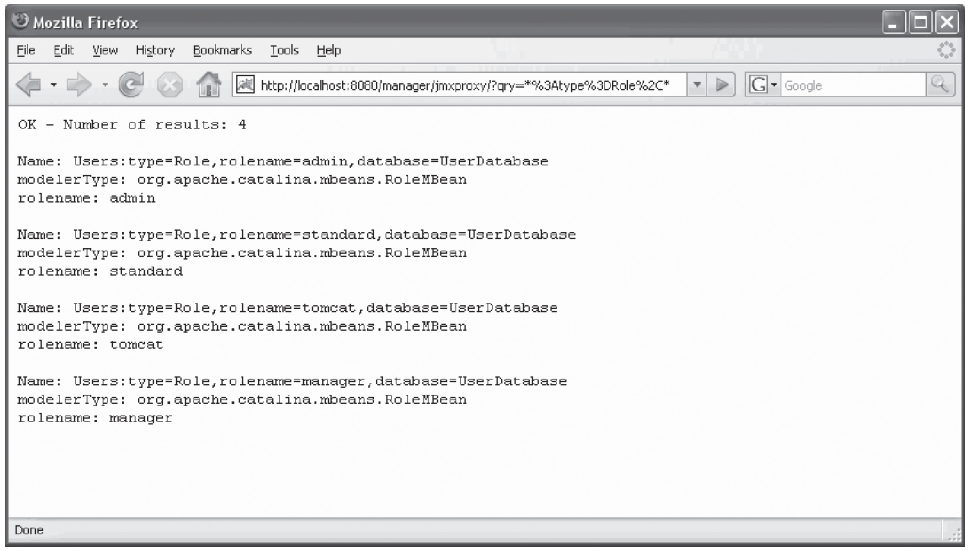

Figure 6-7: Lifecycle listener–created role-type MBeans

Note that you will have to authenticate to use the `manager` *application. This means that the manager role must be added to the* `tomcat-users.xml` *file (either manually or via the* `admin` *application) for the* `manager` *application user. In this test case, the manager role was added to the default Tomcat user. (Chapter 8 discusses the* `manager` *application.)*

Removing Default Lifecycle Listeners

Next, the lifecycle listeners will be removed. As a result, it is expected that the dynamically created MBeans will no longer be available. To try this out, first stop Tomcat 6. Edit the `server.xml` file by hand and comment out the two `<Listener>` elements. The following code shows the elements commented out:

```
<Server port='8005' shutdown='SHUTDOWN' debug='0'>
<!--
 <Listener className='org.apache.catalina.mbeans.ServerLifecycleListener'
           debug='0'/>
  <Listener className='org.apache.catalina.mbeans.GlobalResourcesLifecycleListener'
           debug='0'/>
-->
```

This means that no lifecycle listener will be configured for the server component, and no role-typed MBeans will be created.

Start Tomcat 6 and try the previous proxy URL again. This time, no role-typed MBean will be found, as shown in Figure 6-8.

Because the lifecycle listeners responsible for creating the role-typed MBeans are not configured, the query reveals that no role-typed MBeans are available.

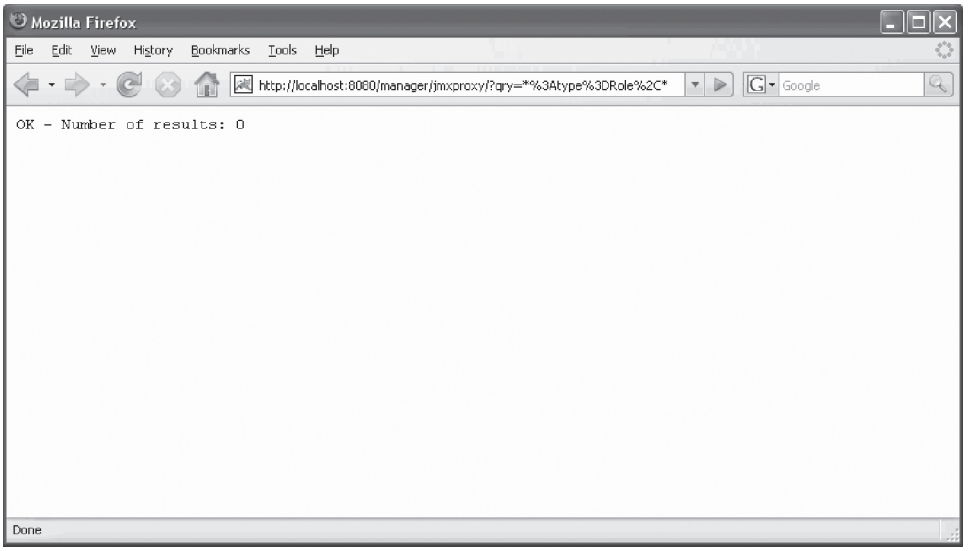

Figure 6-8: No role-typed MBean is available when lifecycle listeners are disabled.

Adding APR Lifecycle Listener and Native SSL Engine Configuration

Apache Portable Runtime (APR) is a platform-specific binary library that Tomcat 6 can make use of to boost its Web server performance.

The following line in `server.xml` loads the APR support class, initializes the APR library, and turns on its SSL engine:

```
    <Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
```

There exists a single configurable attribute associated with the `AprLifecycleListener` in Tomcat 6, called `SSLEngine`. The default value of `SSLEngine`, if not specified, is `on`. The following table summarizes the listener configuration.

Attribute	Description	Required?
`className`	A listener class that loads and initializes the APR library. This is set to `org.apache.catalina.core.AprLifecycleListener`.	Yes
`SSLEngine`	Can be set to `on`, `off`, or a specific SSL engine name supported by the APR library. Default is `on`. This attribute can be useful if you are using OpenSSL native code–based SSL implementation and/or a hardware-accelerated SSL solution.	No

In addition to initialization, the `AprLifecycleListener` listener also intercepts the "after stop" event of the server component and calls the `terminate` method of the APR library to give it a chance to clean itself up.

The actual use of APR requires the download and installation of a native APR DLL. Configuration of the `AprLifecycleListener` will cause Tomcat 6 to attempt to find this DLL. However, if the DLL does not exist, Tomcat 6 will still be functional, but without the APR optimizations.

Initialization of the JSP Processor

The JSP processor in Tomcat 6 is initialized by the following line in the `server.xml` file:

```
    <Listener className="org.apache.catalina.core.JasperListener" />
```

This JSP processor engine is a new engine that supports JSP 2.0 specification and its code name is Jasper 2.

The previous listener configuration has no attributes, and simply provides an early initialization opportunity for the JSP processor prior to loading of any Web applications. Early initialization tasks include setting up a security management hook, and the initialization of an optional pool for JSP (configurable via the `org.apache.jasper.runtime.JspFactoryImpl.USE_POOL` system property, shown in Chapter 21).

The actual JSP processor is a servlet, `org.apache.jasper.servlet.JspServlet`. This servlet is configured in the Tomcat 6 global deployment descriptor — `web.xml` — under the `conf` directory.

Configuration of this JSP processor servlet is discussed in Chapter 5 where the system global `web.xml` is explored.

Summary

This chapter discussed Tomcat configuration topics that are beyond the basic "up-and-running" requirements. The following important areas were covered:

- The Access Log Valve can enable logging of resource access at different levels: the Web application, the virtual host, or globally across all the virtual hosts. This Valve is highly configurable, and you can customize the name as well as the actual format of the log entries, although the `common` format is the best known.
- The standard Single Sign-on Valve enhances the user experience because users no longer must type in a username and password every time they switch between Web applications running on the same host. This Valve caches the credentials on the server and passes them between the applications as required.
- The Request Filter Valves are easily configured to control all incoming requests that are to be processed or blocked entirely. These valves can block a list of IP addresses or host names.
- The lesser-known Request Dumper Valve can be used to debug other valves and/or components, and to visualize the effects of scoping.
- The configurable Persistent Session Manager component can be used to provide a measure of reliability to Tomcat. It can periodically back up sessions on disk, and also swap out dormant sessions to make room for active sections. Most important, it will restore sessions from disk when it starts up. This enables sessions to persist between restarts of the Tomcat server.

- JNDI provides a uniform interface to different directory services. This makes it possible to write only one set of lookup code across different directory services. Examples presented included the configuration of JNDI resources (such as JDBC connections and JavaMail sessions).
- Lifecycle listeners are Java code modules and are configured as XML subelements of a Tomcat component. Configured listeners are invoked by the component during well-defined points in the lifecycle of a component. In Tomcat 6, lifecycle listeners are used to create manageable objects (MBeans) for supporting Tomcat manageability (via JMX), to turn on and off native code implementation of the SSL engine via APR, and to initialize the Jasper 2 JSP processor.

Chapter 7 discusses the configuration for Web applications installed in the Tomcat container.

7

Web Application Configuration

Web applications consist of static content (such as HTML pages and images files) as well as dynamic content (such as servlets, JSPs, and Java classes). Chapter 2 briefly discussed servlets and JSPs.

Although these Web applications usually are created by developers, they often require a system administrator to configure and deploy them, especially if the deployment is on a production machine. A systems administrator needs to know about a number of things in order to administer Web applications, such as the structure of a Web application and its configuration files.

This chapter describes the configuration-related issues for Web applications:

- The structure and content of a Web application
- The deployment descriptor for a Web application (that is, the `web.xml` configuration file)

Chapter 8 discusses other administrative activities for Web applications (for example, deploying, undeploying, and listing Web applications).

Understanding the Contents of a Web Application

Web applications are usually installed under the `<TOMCAT_HOME>/webapps` directory. The Servlet 2.5 specification requires that a certain basic directory structure be followed. Figure 7-1 shows a sample Web application structure.

The Web application is typically deployed in a directory named after the Web application. This name is also used in the Web application URL. For example, the sample Web application in Figure 7-1 is

located in a directory called `exampleapp`, and can be accessed by the URL `http://localhost:8080/exampleapp/`. Here, `/exampleapp/` is called the *context path* for the Web application. The context path refers to everything in the URL after the server and port number, and is the part of the URL that is used to resolve the location of the resource.

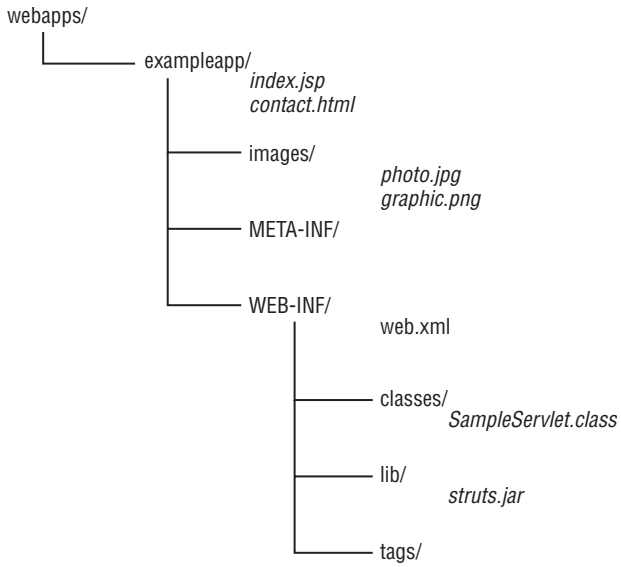

Figure 7-1: Directory structure for a sample Web application

An exception to this is the `ROOT` Web application, which is installed under `<TOMCAT_HOME>/webapps/ROOT`. No context path needs to be specified for the `ROOT` Web application, as shown in the following URL:

```
http://localhost:8080/
```

In terms of the structure of the Web application, the minimum that is required is a `WEB-INF` directory with a `web.xml` file in it. As discussed previously in Chapter 2, HTML and JSP pages belong to the public resources that a client may request directly. All the contents of the `WEB-INF` and `META-INF` directories fall into the category of an application's private resources, and cannot be accessed directly by client applications.

Public Resources

Everything outside the `WEB-INF` and `META-INF` directories are *public resources*, and can be accessed via an appropriate URL. For example, the `contact.html` file can be accessed as follows:

```
http://localhost:8080/exampleapp/contact.html
```

The placement of publicly accessible files (such as JSP and HTML pages, CSS, and images) is arbitrary as far as the specifications for Web applications are concerned, and they can be accessed directly by a client.

> **By *arbitrary*, we do not mean that they can be placed anywhere and the server will find them. Rather, as long as the files are put within the Web application directory, and outside of the `WEB-INF` directory, then the application itself (and its designer) decides where files are placed.**

In the example Web application shown in Figure 7-1, `index.jsp` is the default welcome page for the Web application. The *welcome page* is the Web page served up when you access the Web application URL — in this case, `http://localhost:8080/exampleapp/`.

If this Web page were not present, then, by default, `index.html` and `index.htm` are looked for and served. These welcome pages are subject to configuration and can be modified, as you will see later in the chapter. Besides `index.jsp` and `contact.html`, the other public resources in the example application are the image files in the `images` directory.

URL Mappings

In most cases, when you request a Web resource from your browser (such as an HTML page), it is served to you without modification by the Web server. JSP pages are an exception to this. A JSP page is first passed through a JSP compiler that compiles the file to a Java file, and then compiles the Java file to a Servlet class. This Servlet class then executes, and the output is displayed on your browser.

The code that makes this happen is a URL mapping defined using a `<servlet-mapping>` element, as shown next. This URL mapping is defined in `<TOMCAT_HOME>/conf/web.xml`. This file is the deployment descriptor for all the Web applications — individual Web applications can define their own deployment descriptors.

```
<servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
```

The preceding code specifies that any URL that ends in `.jsp` should be passed to a servlet named `jsp` that is defined elsewhere in the same `<TOMCAT_HOME>/conf/web.xml` configuration file. The definition for this servlet is as follows:

```
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
  ...
  <load-on-startup>3</load-on-startup>
</servlet>
```

As you can see, the fully qualified name of the servlet is `org.apache.jasper.servlet.JspServlet`. The servlet is handed the request, uses the context path to load the JSP page, and passes it to Tomcat's JSP compiler, known as Jasper. The `load-on-startup` option ensures that the Servlet class is loaded into memory on startup with a priority of 3 (where 1 is most important) to ensure that it is loaded before any JSP pages are requested. You'll see other configuration options in the `web.xml`, too, and these are explained in greater detail later in this chapter.

The WEB-INF Directory

The contents of the `WEB-INF` directory are also shown Figure 7-1. As shown, it has a deployment descriptor (`web.xml`) and three subdirectories. These subdirectories include the following:

- The `classes` directory
- The `lib` directory
- The `tags` directory

The classes Directory

The `classes` directory contains servlet and utility classes, including JavaBeans. It may also contain a number of resource files such as key/value message lists, which contain error messages and user prompts for the application, and application-specific configuration information.

Each class is stored within a directory hierarchy that matches its fully qualified name (FQN). Therefore, a class with package structure `com.wrox.db.DatabaseServlet` will be stored in the `classes/com/wrox/db` directory structure. Because servlets are merely Java classes that implement a specified interface, they are stored in the `classes` directory, too. Previously, it was common to place servlets in an additional directory within the `WEB-INF` directory named `servlets`. Classes placed into this directory are no longer on the class path by default, and they need to be moved into the `classes` directory.

Ideally, an administrator need not be concerned with the contents of the `classes` directory. However, it is worth noting that configuration files may be present in it. The resource files mentioned earlier may be within this directory and are typically text files that contain configuration information or are used to externalize error messages. This is merely a programming practice, and you may have any kind of file here.

For example, there may be an `ApplicationResources.properties` file (the name is determined by the application developer) that looks like the following:

```
prompt.username=User Name (your email address)
prompt.password=Please enter you password
error.password.mismatch=The password is incorrect. Please try again.
```

This type of list enables an application developer to refer to the text by its name (for example, `prompt.username`), thereby enabling an administrator to change the values, minimizing the need to touch the sensitive JSP code.

The Java classes and property files placed in the `classes` directory are accessible only by that Web application. Earlier Tomcat versions allowed classes shared across web applications to be placed in `<TOMCAT_HOME>/shared/classes`, but this is no longer the case.

The tags Directory

An optional `tags` directory within the `WEB-INF` directory contains configuration files for tag libraries.

A *tag library* is a group of Java classes that define the functionality of dynamic markup tags. For example, you can use a tag that you define as follows:

```
<date:today/>
```

This would output the current date whenever it is placed in a JSP file. To enable the container to recognize which Java class to invoke when it comes across the tag, you must provide a configuration file that lists the number of arguments the tag can have, its name (in this case, the tag's name is `today` and the library it belongs to is `date`). The tag library configuration files have a `.tld` extension. The configuration of a tag library is the territory of developers and designers, and thus beyond the scope of this book.

The lib Directory

This directory contains packaged Java libraries (`.jar` files) that the application requires and that are bundled with the application. JAR files that are placed here are available only to the Web application. If the libraries are to be accessed across Web applications, they should be placed under `<TOMCAT_HOME>/lib`. This is different from earlier Tomcat versions, which had a `<TOMCAT_HOME>/shared/lib` directory for libraries that needed to be shared with other Web applications.

The following section describes what aspects of the `web.xml` configuration file you can administer.

The META-INF Directory

As mentioned, the `WEB-INF` directory represents the private resources of an application. However, this is not the only directory for private resources. A Web application may have an optional `META-INF` directory that contains deployment information for tools that create `war` files and resources that applications may rely on. Therefore, a Servlet container will refuse to show the contents of the `META-INF` directory to a client.

The `META-INF` directory can contain two configuration files: the manifest file (`MANIFEST.MF`) and the context file (`context.xml`).

The Manifest File

The `MANIFEST.MF` file is an optional configuration file for a Web application. It contains a list of JAR files on which an application relies. The container can then use this to check for all the required libraries that are to be made available for the Web application.

An entry in this text file should be provided as follows, on a single line:

```
Extension-List: extension1 extension2 extension3
```

Each extension name is separated by a space and is placed as a separate entry in the `MANIFEST.MF` file. The entries are named with a prefix, followed by the string `-Extension-Name`, which is an attribute name, as shown here:

```
extension1-Extension-Name: com.wrox.extension1
extension1-Specification-Version: 1.0
extension1-Implementation-Version: 0.8
extension1-Implementation-Vendor: WROX Press Ltd
extension1-Implementation-Vendor-Id: com.wrox
extension1-Implementation-URL: http://www.wrox.com/extension1/
```

As you can see, the name of the extension is referenced in each entry. This is suffixed by a specific attribute name describing the extension. The name of the extension in this file is an alias for the extension's name as defined in the `jar` file. Thus, the declaration of the extension's alias is accomplished by simply prefixing it to the attribute names; it does not need to be explicitly defined.

The extension's proper name is referred to in the first entry. The server will investigate the contents of each `jar` file installed on it and check packages to determine whether the names match. The specifications and implementation version numbers are self-explanatory, as should be the vendor name that is specified in the `Implementation-Vendor` attribute. The vendor should be a globally unique ID. The custom of including the reversed host name is common. In the preceding example, this is `com.wrox`.

Finally, the `Implementation-URL` should be provided, giving the location of additional information and often download instructions. For our purpose, this is the most useful line. If the extension is not installed, the URL should provide enough information to ensure that it is made available to the Web application by other means.

The manifest file is typically generated automatically when a Web application is packaged as a Web archive (`.war`) file. Packaging Web applications for distribution as `.war` files is described in Chapter 2.

The Context File

The context file (`context.xml`) contains the configuration for the Web applications' `Context`. As you saw in Chapter 4, the `Context` represents a Web application, and has all the configurable elements for it.

This is not the only place where the Web applications' `Context` is defined: Chapter 5 describes this in greater detail, and lists all the places where the Web application `Context` can be configured.

Understanding the Deployment Descriptor (web.xml)

A *deployment descriptor* is an XML file that contains configuration information used by the Web application for execution on the Servlet engine. The deployment descriptor for a Web application is the `<TOMCAT_HOME>/webapps/<webapp name>/WEB-INF/web.xml` file. There is another `web.xml` file that is applicable for all Web applications deployed in the servlet Engine, and this is located under `<TOMCAT_HOME>/conf`. This section examines application-specific deployment only. However, the configuration-related information is valid for all deployment descriptors.

The Servlet 2.5 specification uses a schema for the deployment descriptor — older specifications (Servlet 2.3 and earlier) still use a Document Type Definition (DTD). Both are still supported for backward compatibility with existing Web applications. Because you might need to support existing Web applications, this chapter covers both the older Servlet 2.3–style `web.xml` and the schema-based versions. The first few lines of the deployment descriptor indicate whether it is the Servlet 2.4/2.5 schema-based `web.xml` or the older DTD-based version.

Following is a Servlet 2.3 DTD-based `web.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
   ...
</web-app>
```

Following is a Servlet 2.4 schema-based `web.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
         version="2.4">
  ...
</web-app>
```

Finally, the new Servlet 2.5 schema-based `web.xml` is as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
        version="2.5">
  ...
</web-app>
```

If you use a Servlet 2.3–type deployment descriptor, Tomcat ignores all the new Servlet and JSP features that you use in your application, such as JSP EL (Expression Language).

The Servlet 2.3–Style Deployment Descriptor

The `web.xml` file takes the following generalized form:

```
<?xml version="1.0"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
  <icon>
  <display-name>
  <description>
  <distributable>
  <context-param>
  <filter>
  <filter-mapping>
  <listener>
```

(continued)

```
    <servlet>
    <servlet-mapping>
    <session-config>
    <mime-mapping>
    <welcome-file-list>
    <error-page>
    <taglib>
    <resource-env-ref>
    <resource-ref>
    <security-constraint>
    <login-config>
    <security-role>
    <env-entry>
    <ejb-ref>
    <ejb-local-ref>
  </web-app>
```

The order of elements inside the `<web-app>` element must be as shown previously, but some elements are optional, and others may appear multiple times. The following table may be used as a quick reference to the functionality of each element. A more detailed explanation is provided later in the chapter.

Element	Description	How Many?
`<icon>`	Image for an application	0 or 1
`<display-name>`	Display name for a Web application	0 or 1
`<description>`	Description used for display	0 or 1
`<distributable>`	A Boolean value indicating whether an application is distributable across servers	0 or 1
`<context-param>`	Initialization parameters for the entire application	0 or more
`<filter>`	Defines a filter Valve	0 or more
`<filter-mapping>`	Defines a URL pattern to which the given filter needs to be applied	0 or more
`<listener>`	Defines a lifecycle event listener	0 or more
`<servlet>`	Defines a servlet	0 or more
`<servlet-mapping>`	Defines a URL pattern to invoke a named servlet	0 or more
`<session-config>`	Defines session configuration	0 or 1
`<mime-mapping>`	Defines the MIME type for a given file type	0 or more
`<welcome-file-list>`	A list of files to be served if no resource is specified explicitly in the URL	0 or 1
`<error-page>`	Defines a Java exception or an HTTP code-based error page	0 or more
`<taglib>`	Declares a tag library	0 or more
`<resource-env-ref>`	Declares a resource-administered object	0 or more
`<retsource-ref>`	Declares an external resource	0 or more

Element	Description	How Many?
`<security-constraint>`	Restricts access to a resource to a required transport guarantee and by user role	0 or more
`<login-config>`	Defines authentication parameters	0 or 1
`<security-role>`	Declares a security role by name	0 or more
`<env-entry>`	Defines a Web application's environment entry	0 or more
`<ejb-ref>`	Declares a reference to an EJB's home	0 or more
`<ejb-local-ref>`	Declares a reference to an EJB's local home	0 or more

In the following sections, you examine a minimal `web.xml` file to understand what must be present.

The XML Header

Every `web.xml` file complies with the XML specifications that require an XML header in the beginning of the file, as shown here:

```
<?xml version="1.0"?>
```

Optionally, the declaration may also include an encoding type that identifies the character encoding of the document, as is standard for XML. For example, if the document is encoded in UTF-8, the declaration may be provided as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
```

The DTD Declaration

The next tag is a Document Type Definition (DTD) tag. A DTD is a document that outlines the structure of the `web.xml` elements, what elements are allowed and in which order, and their content. The inclusion of a standard DTD declaration in our `web.xml` file looks as follows:

```
<?xml version="1.0"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

Applications that comply with the Servlet specifications prior to 2.3, such as Tomcat 3 `web.xml` files, for example, will have the following DTD reference:

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

Backward compatibility is required as per the Servlet 2.3 specifications, so applications that were written for the Servlet 2.2 specifications will work unaltered on Tomcat 6, except for any dependencies on the exact configuration of the server (such as the location of databases, network authentication, and the name of the server and the host). Because the Servlet 2.3 specifications have introduced a number of new tags since 2.2, we will also highlight these tags where appropriate.

<web-app>

The root element of the `web.xml` file is `<web-app>`, and all other XML elements reside inside it:

```
<?xml version="1.0"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
</web-app>
```

This is all that is required for the `web.xml` file to be complete. However, in many practical cases, there will be more. Let's begin by covering elements that describe the application. A number of elements are provided so that deployment tools can identify Web applications visually and textually.

<icon>

This tag holds the location of the image files within the Web application that may be used by a tool to represent the Web application visually. The `<icon>` tag may contain two child elements (`<small-icon>` and `<large-icon>`) that carry the location of a 16 × 16–pixel image file and a 32 × 32–pixel image file, respectively.

```
<icon>
  <small-icon>/images/icons/exampleapp_small.gif</small-icon>
  <large-icon>/images/icons/exampleapp_large.gif</large-icon>
</icon>
```

<display-name>

This tag provides a name that can be used for display in a GUI. The name need not be unique. For example, the following display name is typical:

```
display-name>Example Application</display-name>
```

<description>

This tag contains the description of a Web application, as shown in the following example:

```
<?xml version="1.0"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <icon>
    <small-icon>/images/icons/exampleapp_small.gif</small-icon>
    <large-icon>/images/icons/exampleapp_large.gif</large-icon>
  </icon>

  <display-name>Wrox Example Application</display-name>

  <description>
    The Wrox example application contains a number of simple resources
    for illustrating configuration points.
  </description>
</web-app>
```

These element tags must be listed in the same order as shown earlier in the section (refer to `http://java.sun.com/dtd/web-app_2_3.dtd` for more information). The actual number of tools for deploying Web archives (especially in a drag-and-drop manner as suggested by the use of icon files) is somewhat low, so it is common for these values not to be provided. The `web.xml` file may be heavily commented. XML comments take the same form as HTML ones:

```
<!--
This is a comment
-->
```

<distributable>

This tag describes a Web application that is designed to be distributable for load balancing and fail-over. By default, the value of this is `false`.

<context-param>

Context parameters are mechanisms used for setting application-initialization parameters. For example, you could set the URL to a database here. The following example enables the administrator to change the title and greeting of the example application:

```
<context-param>
  <param-name>title</param-name>
  <param-value>Wrox example application - Chapter 7</param-value>
</context-param>
<context-param>
  <param-name>greeting</param-name>
  <param-value>Welcome to the example application</param-value>
</context-param>
```

There may be any number of context parameters in the application, known as *initial parameters*. Each dynamic resource (such as a servlet, a JSP page, or a class) with access to the application context is able to look up the value associated with a given parameter name. Typical items provided as a context parameter are the debug status of the application, the verbosity of logging (these two are often interlinked), and as much other externalized configuration as the application developer has allowed.

<filter>

Filters are new to the Servlet 2.3 specifications. Filters are reusable components that intercept the client request and response and apply some type of processing to them. For example, a filter may apply compression to the contents of the response, thus reducing bandwidth usage and improving the performance of the application by minimizing the size of the response packet. This is just an example, of course, and to make it work in a real-world situation would require additional support in the browser.

Filters are intended to be the ultimate reusable Web components. They should be virtually independent of the content being created. Examples include the compression filter, a transformation filter that may convert XML to HTML or WML, a filter to provide the logging of resource usage, and a filter to restrict access to resources.

A filter, like all Web application resources, can be mapped to a URL pattern, including the extension of the resource, a section of the site (such as everything within the `images` directory), or even a URL alias such as the `servlet` alias that exists on most default installations of Java Web servers.

In addition, filters can have an icon associated with them, and configuration parameters (initialization parameters). An example configuration is shown here:

```
<filter>
  <icon>/images/icons/filter.jpg</icon>
  <filter-name>Compressor</filter-name>
  <display-name>Compression Filter</display-name>
  <description>This filter applies compression</description>
  <filter-class>com.wrox.utils.CompressionFilter</filter-class>
  <init-param>
    <param-name>compression_type</param-name>
    <param-value>gzip</param-value>
  </init-param>
</filter>
```

Once a filter is defined, it can be mapped against any number of URL patterns. In addition, when many filters are defined for a given URL pattern, they are all applied in the order in which they are defined in the `web.xml` file. In the following example, the compression filter is applied to every URL:

```
<filter-mapping>
  <filter-name>Compressor</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

Filters are covered in Chapter 6.

<listener>

Listeners are designed to respond to events in an application. For example, a JavaBean could send an e-mail when an event requiring administration is recorded:

```
<listener>
  <listener-class>com.wrox.listeners.ExampleListener</listener-class>
</listener>
```

Listeners are also covered in Chapter 6.

<servlet>

A servlet is declared in the `web.xml` file by assigning it a unique name, which references its fully qualified name against a shorter, more intuitive name:

```
<servlet>
  <icon>/images/icons/DownloadServlet.jpg</icon>
  <servlet-name>Download</servlet-name>
  <display-name>File Download Servlet</display-name>
  <description>
    This Servlet manages file downloads in the application
  </description>
  <servlet-class>com.wrox.servlets.DownloadServlet</servlet-class>
  <!-- require terms and conditions agreement? -->
  <init-param>
    <param-name>require_tc</param-name>
```

```
      <param-value>true</param-value>
    </init-param>
    <load-on-startup>5</load-on-startup>
    <!-- uncomment this if Servlets must run in user role
    <run-as>
      <description>
        This Servlet does not require authorization to resources
      </description>
      <role-name>admin</role-name>
    </run-as>
    -->
  </servlet>
```

In the preceding example, the servlet manages the download process, enabling you to decide at runtime if a user is required to sign a terms and conditions acceptance form before download commences. The optional `<icon>`, `<display-name>`, and `<description>` elements work in the same way as described previously. The fully qualified name of the servlet is specified in the `<servlet-class>` element.

Because JSP pages are ultimately compiled into servlets, an alternative to the servlet class name (`<servlet-class>` element) is to specify the JSP file name (`<jsp-file>` element) to which these configuration parameters should be applied, thus making JSP files fully configurable. The reference is a full path, from the root of the application to the JSP file, as shown in the following example:

```
  <servlet>
    <servlet-name>ExampleJSP</servlet-name>
    <jsp-file>/admin/users/ListAllUsers.jsp</jsp-file>
    <!-- list disabled user accounts -->
    <init-param>
      <param-name>list_disabled_accs</param-name>
      <param-value>false</param-value>
    </init-param>
  </servlet>
```

The initialization parameters work in the same way as the application context parameters do. However, they are specific to the servlet.

The `<load-on-startup>` element specifies an integer value indicating whether the servlet must be loaded when the Tomcat server boots, rather than on a client's request. If this value is not specified or it is negative, the container loads the servlet into memory when the first request comes in.

If the value is zero or a positive integer, the container must load the servlet into memory at startup of the Web application. Servlets assigned lower integers are loaded before those with higher integers. Servlets with the same `<load-on-startup>` values are loaded in an arbitrary sequence by the container.

In the `Download Servlet` example, the `<run-as>` attribute is not specified because it is commented out. However, if the servlet requires a privileged role, it can be specified here, so that any resource requiring a privileged user will discover it while calling the `isUserInRole()` method.

<session-config>

Session configuration enables sessions to be configured for every application. The `<session-timeout>` element can be used to set a session timeout value. This value can be calculated by considering typical

client usage patterns, along with security requirements. For example, if a user is asked to enter a great deal of information, the session timeout value may be set to a larger number to avoid information being lost.

Alternatively, in low-security environments with serializable sessions, it is possible to set sessions to never expire so that the user is always recognized.

The session configuration is defined as follows:

```
<session-config>
  <session-timeout>40</session-timeout>
</session-config>
```

If the value is zero or less, the session is never expired and the application must explicitly remove it as required. If the element is not provided, the default value of 30 is used as specified in the global `web.xml` file within Tomcat's `<TOMCAT_HOME>/conf` directory.

<mime-mapping>

MIME types enable browsers to recognize the file type of the content being returned by the server so that the browser can handle it correctly, to determine whether to display it (as HTML, plain text, images); pass the content to a plug-in (such as Flash); or prompt the user to save the file locally.

Tomcat comes preconfigured with the majority of MIME mappings set, which can be seen in the `<TOMCAT_HOME>/conf/web.xml` file. MIME mappings set in this file apply to all applications. Additional MIME mappings may be configured on each Web application with the `<mime-mapping>` element. This can be especially useful when the developer defines new extensions to suit the application. In addition, this can be useful if you wish to have a certain MIME type treated differently from the way it would normally be treated. For example, for a content management application, you may want to prevent Internet Explorer from recognizing the MIME type and thus opening the file in the appropriate application, and instead prompt the user with the File Save dialog box.

Another example might be the automatic generation of Excel files. Excel will accept comma-separated values and convert them to an Excel spreadsheet if the MIME type sent to Internet Explorer is set to the Excel MIME type of `application/x-excel` or `application/ms-excel`. This will open Excel, although the file is a CSV file. This technique is used in Web applications for non-integrated applications in which a company administrator wants to be able to dynamically generate Excel files from a site into their reports, because creating Excel sheets on-the-fly is quite complex.

> **For those interested in creating Excel sheets or manipulating documents on-the-fly, a number of programs can be used for this, such as JExcel (`http://jexcelapi.sourceforge.net/`) or Jakarta POI (`http://jakarta.apache.org/poi/index.html`).**

This is a common technique when it is desirable to use an external application to view content from a Web application/script. The following example shows how the Excel-CSV MIME mapping is done:

```
<mime-mapping>
  <extension>csv</extension>
  <mime-type>application/x-msexcel</mime-type>
</mime-mapping>
...
```

<welcome-file-list>

Sometimes a request is made from a client to an application without a definite resource specified in the URL. For example, the root of the application is requested as follows:

```
http://localhost:8080/exampleapp/
```

whereas a definite resource is requested as shown in the following URL:

```
http://localhost:8080/exampleapp/whatsnew.jsp
```

In such cases, the Web application looks for a file called `index.jsp` in the Web application's directory and executes this file if it exists. If this file cannot be found, it looks for `index.htm` and `index.html` in turn. This is because the welcome file list defined in `<TOMCAT_HOME>/conf/web.xml` includes these files by default. If the `web.xml` file in the `WEB-INF` directory of your Web application does not mention a welcome file list, the default is used.

The format for the welcome file list is as follows (this will apply to each request that does not specify a resource). This means that each of the subdirectories within the application root will also have this rule applied to it. In the following example, `default.jsp` will be loaded instead of `index.jsp`:

```
<welcome-file-list>
  <welcome-file>default.jsp</welcome-file>
  <welcome-file>default.htm</welcome-file>
  <welcome-file>UserWelcome.jsp</welcome-file>
</welcome-file-list>
```

Note that if none of the files in the example list are found, then, depending on the configuration, an HTTP 404–Not Found error message is displayed.

<error-page>

The default behavior for Web applications written in JSP is to return a *stack trace*, which is a complex view into the internals of the virtual machine that greatly reduces the user-friendliness of the application.

You can configure error pages to provide a user-friendly mechanism for informing users about a problem, enabling them to continue using the application. The errors are mapped to the HTTP specification error mappings (such as a code for a resource that cannot be found, server malfunctioning, authentication issues, resource issues, and so on).

In addition, because there are no one-to-one correspondences between HTTP errors and Java exceptions, the exception class type may be specified. This enables the creation of error pages that are generic, and follows good programming practice. Someone without an understanding of the application's internals can configure them. Following is an example for the common 404 message for a `NullPointerException`:

```
<error-page>
  <error-code>404</error-code>
  <location>/errors/oops.jsp</location>
</error-page>
<error-page>
  <exception-type>java.lang.NullPointerException</exception-type>
  <location>/errors/badlycodedpage.jsp</location>
</error-page>
```

Like the JSP page example, `<location>` must be a reference from the root of the application.

These pages often have a message that notifies the user of the problem, and provisionally provides both a search box so that users can locate the resources they need and a list of likely links in the site from which they might receive help.

Often the problem is a configuration issue, and users are best served by being informed that the problem will be fixed and they should return later. The developer may be informed through automated parsing of error logs or through a notification system that sends e-mails to a watched e-mail address or directly to the administrator or the development team.

Should any problem occur in a page (such as missing resources, a bug in the software, or parts of the system being down), a page configured here would be returned. Error pages can also be written so that they display contextual information (that relates to the specific problem at hand), but this requires an understanding of the inner workings of the system and can only be provided by a developer.

HTTP return codes can be found at the following URL:

```
www.w3c.org/Protocols/HTTP/HTRESP.html
```

Error pages are configured by associating them with the HTTP return code that covers the error group. Two examples are provided in the following example, one for the `HTTP 404` code and one for a `NullPointerException` (an internal error that is often hard to debug in an application and may require a developer's intervention to correct):

```
<error-page>
  <error-code>404</error-code>
  <location>/errors/ResourceNotFound.htm</location>
</error-page>
<error-page>
  <exception-type>java.lang.NullPointerException</exception-type>
  <location>/errors/ApplicationProblem.jsp</location>
</error-page>
```

<taglib>

Tag libraries, as previously discussed in Chapter 2, are reusable Java components that may be invoked using markup tags in the page. The tag library definition is specified by the application developer and the HTML designers. However, the main configuration of these reusable components is done in a separate file (one with a `.tld` extension), as this entry simply enables aliasing of the location of this configuration document against a URI. The exact location of the configuration file, which is given as a reference to the file from the Web application's root directory, can then be referred to by its alias.

This aliasing enables location-independence (that is, the tag library configuration files can be moved around without editing the JSP pages that refer to the tag library configuration file, so long as the tag entry points to it). An example entry is shown here:

```
<taglib>
  <taglib-uri>applicationtags.tld</taglib-uri>
  <tablib-location>/WEB-INF/tlds/web-app.tld</taglib-location>
</taglib>
```

In this example, the tag library configuration file that the Web application container needs for resolving references, looking up initialization parameters, and enforcing proper use of the tags is referred to by

its alias, `applicationtags.tld`. The location of the configuration file is customarily within the `WEB-INF` directory in a directory called `tlds`. If this location is changed, you must adjust the `<taglib-location>` entry, but any code referencing it can stay the same.

<resource-ref>

Two elements, `<resource-ref>` and `<resource-env-ref>`, are provided for configuring resources for a Web application environment. These elements enable two things:

- **The management of connections to resources, such as a reference to the object pooling resource connection (much like database-connection pooling), to make the process more efficient:** Object pooling enables efficient use of resources by defining a component that manages connections to those resources. In the case of databases, the pool will make a number of connections and when a client requests one, it is handed over to the client to be used. When the client requests the connection to be closed, the pool retrieves the connection, but rather than closing it and establishing a new connection, it reuses the connection by handing it over to the next client (as long as the authority constraints and the type of connection matches).

  Because establishing a connection to a database is a resource-intensive process, this can affect application performance. A pool can also be configured to refresh the connections periodically and to restore dropped connections so that the application can efficiently recover from database failures.

- **A reference to administered objects, which provides the application with access to runtime administration of the resource:** Administered objects enable the application configuration to be changed without restarting the server. They can also be used to monitor the state of the application by interrogating administered objects for their current state.

<security-constraint>

Web resources may be associated with some security constraints for authentication purposes. The constraints limit access to the resource according to user roles (such as manager, administrator, user, and guest) and by transport guarantee (which can include SSL secure data transmission), guaranteeing delivery and non-interference.

The `<security-constraint>` element enables a Web resource to be defined against an authentication constraint and a user data constraint.

An entry takes the following form:

```
<security-constraint>
  <display-name>Name String</display-name>
  <web-resource-collection>
    <web-resource-name>GETServlet</web-resource-name>
    <description>
      Group together all Servlet GET requests on the server using
      /servlet/servletname. We are grouping these requests as (we have
      decided) they require additional security being inherently less secure
      than the POST method and aliased Servlet calls.
    </description
    <url-pattern>/servlet/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
```

(continued)

```
      <description>
         All roles are constrained to secure connection to Servlet resource
         via GET calls
      </description>
      <role-name>*</role-name>
    </auth-constraint>
    <user-data-constraint>
      <description>
         Constrain the user data transport for GET Servlet requests to secure
         sockets
      </description>
      <transport-guarantee>INTEGRAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
```

New in Servlet 2.5: *While Servlet 2.4 allowed only the standard set of HTTP methods (*`GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`, *and* `TRACE`*), Servlet 2.5 allows extension HTTP methods, including the Web-based Distributed Authoring and Versioning (WebDAV) methods (*`LOCK`, `UNLOCK`, `COPY`, *and* `MOVE`*).*

<web-resource-collection>

The `<web-resource-collection>` element identifies a group of resources and the methods by which these resources can be requested. In the previous example, all Servlets identified by the pattern `/servlet` can be accessed via the HTTP `GET` method. Any number of URL patterns and valid HTTP methods may be provided to exactly define the resource collection.

<auth-constraint>

The `<auth-constraint>` element uses role-based authentication to constrain access to Web resources. You can limit groups of users to whom this security constraint is applied to using role-based authentication. Therefore, placing `administrator` in the `<role-name>` tag — for example, as `<role-name>administrator</role-name>` — allows only users belonging to that role to access the servlets. Role-based authentication is discussed in more detail in Chapter 14.

Valid values are specified by the developer of the application. In the preceding example, `<role-name>*</role-name>` indicates that all roles should be allowed access. An empty element indicates that no roles should be allowed access to the resource.

There is no constraint on the number of `<role-name>` elements required to define security constraints. If none are provided, then the resource is unavailable as no authentication is possible. You might make resources unavailable for security reasons by removing all references to `<role-name>` elements in the `web.xml` file and then restarting Tomcat.

<user-data-constraint>

The `<user-data-constraint>` element indicates what guarantees are given about the communication of data from and to the client. A value of `NONE` indicates that no guarantees are provided that the data has not been tampered with or intercepted by anyone other than the client and the system (the server). Conversely, a value of `INTEGRAL` guarantees the authenticity of the data — i.e. that the data has not been interfered with — whereas `CONFIDENTIAL` guarantees that the data has not been intercepted by a third

party. Specifying `INTEGRAL` or `CONFIDENTIAL` means that SSL will be used by redirecting the client to the SSL port of the server.

This type of configuration is likely to be defined at design time. However, in a well-designed application, it is up to the deployment engineer/system administrator to follow the design architecture to enforce the security constraints defined within it. This enables authentication requirements to be absent from the application code itself, thus allowing the application to be very flexible so that it can be configured as business needs dictate.

<login-config>

This element relates to the configuration of login authentication in the application. The `<login-config>` element contains the authentication method, the Realm name and login page, and the authentication error page that should be used if form-based authentication is specified:

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>MemoryRealm</realm-name>
  <form-login-config>
    <form-login-page>login.jsp</form-login-page>
    <form-error-page>notAuthenticated.jsp</form-error-page>
  </form-login-config>
</login-config>
```

The authentication method consists of the HTTP methods available — namely, `BASIC`, `DIGEST`, `FORM`, and `CLIENT-CERT`. These correspond to basic authentication (plain text), digest (base64-encoded responses), `FORM`-based authentication (which enables an HTML page with a form that prompts the user to log in and returns the username and password), and client-certificate–based authentication, respectively.

The `<realm-name>` identifies the Realm that the server should use to authenticate the user against — in our example, the Realm name alludes to the file-based list of users and passwords provided by the memory Realm with Tomcat. In a production environment, using the memory Realm is not recommended. Instead, a JDBC or JDNI Realm is far more robust and maintainable.

Having chosen form-based authentication, we must specify the login page and the error page, in case a login fails. In this case, we have specified that `login.jsp` contains the login request form. Bad authentication requests are redirected to `notAuthenticated.jsp`.

<security-role>

Security roles were discussed briefly earlier in the chapter. The `<security-role>` element enables roles to be defined along with the optional description:

```
<security-role>
  <description>
    Administrator of the application is allowed read/write rights to the
    content
  </description>
  <role-name>administrator</role-name>
</security-role>
```

Further detail is provided in Chapter 14.

<env-entry>

The <env-entry> element is used to declare environment entries. These are JNDI value parameters that can be used to configure the application. Unlike context initialization parameters, these values are dynamic. They can be referred to and updated at runtime so that the application can be reconfigured dynamically and resources outside the Web application can access them. In particular, they can be administered by non-Java application components and can be managed as part of the entire enterprise administration system.

This works like all JNDI resources; the parameter is referenced from the JNDI initial context and can be accessed using the java:comp/env environment naming context. The env-entry is defined as relative to this context.

The environment entry must be typed to a Java data type (such as String or Integer) so that it can be used within the application and can be used to define environment limits (such as minimum and maximum values). The general structure of an environment entry is as follows:

```
<env-entry>
  <description>Lower limit – minimum allowable value</description>
  <env-entry-name>MinimumValue</env-entry-name>
  <env-entry-value>5</env-entry-value>
  <env-entry-type>java.lang.Integer</env-entry-type>
</env-entry>
```

Environment entries are usually specific to the environment in which they are operating (that is, they are application-specific). However, accepted norms for resource naming may be adopted in an attempt to harmonize resource configuration.

The value can then be accessed using code fragments such as the following:

```
/* Create a lock for context retrieval */
private final Object lock=new Object();
...
synchronized (lock) {
  /* Obtain the initial context */
  Context initCtx=new InitialContext();
  Context envCtx=(Context) initCtx.lookup("java:comp/env");
  // Look up environment entry
  Integer minValue=(Integer)envCtx.lookup("MinimumValue");
} /* End lock */
```

In this code example, the call to get the JNDI Initial Context and do a lookup for the java:comp/env context is inside a synchronized block because it is not thread-safe.

The Servlet 2.4/2.5–Style Deployment Descriptor

Because the Servlet 2.5 and 2.5 style deployment descriptors are very similar to each other, they are covered together.

In the web.xml schema, the web-app element is the root element for the deployment descriptor, and all other elements are contained within it. Unlike the DTD-style 2.3 Deployment Descriptor, the enclosed elements can be in any order.

These elements are listed in the following table. Except for `session-config`, `jsp-config`, and `login-config`, all other elements can occur multiple times in the `web.xml` file.

Element Name	Description
`context-param`	Contains the Web application's Servlet context initialization parameters
`description`	Provides a description for the Web application
`display-name`	Specifies a short name for the Web application
`distributable`	Indicates that this Web application is programmed to be deployed in a distributed Servlet container
`ejb-local-ref`	Declares a reference to the Enterprise bean's (EJB) local home
`ejb-ref`	Declares the references to the EJB's home
`env-entry`	Declares the Web application's environment entries
`error-page`	Defines a mapping between an error code or exception and an error page
`filter`	Declares and configures a filter for the Web application
`filter-mapping`	Specifies the filters to be applied to the Web application, and the order in which they are applied
`icon`	Specifies file names for icons used to represent the parent elements
`jsp-config`	Specifies global configuration properties for the JSP pages in the Web application
`listener`	Configures the properties of an application listener bean
`locale-encoding-mapping-list`	Specifies the mapping between locales and their encoding
`login-config`	Specifies the authentication methods to be used for accessing the Web application
`message-destination`	Specifies a message destination
`message-destination-ref`	Contains the deployment component's reference to a message destination
`mime-mapping`	Defines the mapping between an extension and a MIME type
`resource-env-ref`	Contains a reference to an administered object associated with a resource
`resource-ref`	Contains a reference to an external resource
`security-constraint`	Specifies security constraints for one or more groups of Web resources

Table continued on following page

Element Name	Description
`security-role`	Defines the security roles used in the security-constraint element
`service-ref`	Contains the reference to a Web service
`servlet`	Configuration for a servlet
`servlet-mapping`	Specifies the mapping between a servlet and URL pattern
`session-config`	Defines the session parameters for the Web application
`welcome-file-list`	Specifies a list of welcome files for a Web application

The following sections describe these elements in more detail.

> **Some of the deployment descriptor elements deal with configuration for Java EE components, such as Enterprise JavaBeans (EJBs) and Web services. The following sections cover their configuration, but do not explain these components in any detail.**

The Servlet 2.5 specification introduces a few features convenient for mapping servlets and filters; they represent an improvement over those offered in the earlier specification. These new features are indicated with "New in Servlet 2.5" note text in the following sections.

web-app

In the new `web.xml` schema, the `web-app` element is the root element for the deployment descriptor. A sample `web-app` element for Servlet 2.5 is shown here:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
        version="2.5">
     ...
</web-app>
```

New in Servlet 2.5: *The optional* `full` *attribute in the* `web-app` *element.*

The `web-app` element for Servlet 2.5 has an optional attribute called `full`. Setting `full` to `true` (the default is `false`) causes the Servlet container to ignore any annotations in the class files of the Web application. Annotations are new in Java SE 5; they allow Java programs to embed special configuration information, instead of specifying it in configuration files. Details of annotations are not covered here. For more details see *Professional Java JDK 6 Edition* (ISBN 978-0-471-77710-6). While these annotations are a great convenience for developers, they have a performance impact during class loading by the Servlet container. If no annotations are used by the Web application, then the `full` attribute should be used to tell the container not to do these checks while loading up the Java classes.

An example of the `web-app` element with the `full` attribute is shown here.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
        version="2.5"
         full="true">
      ...
</web-app>
```

The `web-app` element for Servlet 2.4 would look like the following:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
         version="2.4">
    ...
</web-app>
```

As mentioned earlier, all other elements in the deployment descriptor are contained within the `web-app` element.

context-param

The `context-param` element contains name-value pairs containing a Web application's Servlet context initialization parameters. The `context-param` has the following subelements:

- `description`: A text description of the name-value pair
- `param-name`: Name of the initialization parameter
- `param-value`: Value of the initialization parameter

A sample usage is shown here:

```
<context-param>
  <param-name>webmaster</param-name>
  <param-value>webmaster@foobar.com</param-value>
  <description>Email address of webmaster</description>
</context-param>
```

description

The `description` element provides a textual description for its parent element. When included under the `web-app` element (as shown in the "JSP examples" section of the Web application included in the Tomcat distribution), it describes the Web application. This element is used elsewhere, too (for example, inside the `context-param` and `filter` elements), where it provides a description for each element:

```
<description>
  JSP 2.0 Examples
</description>
```

The `description` element has an optional attribute, `xml-lang`, which indicates the language of the description text. This defaults to `en` for English. There can also be multiple description elements, usually with different `xml-lang` attributes, to support localization.

display-name

The `display-name` element gives a short, descriptive name for the parent element. For example, when used directly under the `web-app` element, it provides a name for the Web application. This name is displayed by software tools that work with deployment descriptors. Like the `description` element, the `display-name` element too has an `xml-lang` attribute (which defaults to `en`) to indicate the language; and multiple `display-name` elements with different `xml-lang` values can be used to handle multiple-language support. A sample `display-name` element is shown here:

```
<display-name xml-lang="en">JSP 2.0 Examples</display-name>
```

distributable

The presence of a `distributable` element indicates that the Web application has been programmed to be deployed (if required) in a distributed Servlet container. Such a Servlet container may distribute the Web application to multiple JVMs for scalability or performance considerations.

```
<distributable/>
```

Chapter 17 discusses a deployment scenario in which this is used.

ejb-local-ref

The `ejb-local-ref` element declares a reference to the enterprise bean's (EJB) local home. This element has the following child elements:

- `ejb-ref-name`: The EJB reference name
- `ejb-ref-type`: The EJB reference type
- `ejb-link`: Specifies that the EJB reference is linked to an enterprise bean
- `local`: The fully qualified name of the EJB's local interface
- `local-home`: The fully qualified name of the EJB's local home interface

ejb-ref

This element contains a reference to an EJB's home. It has the following child elements:

- `ejb-refname`: The name used in the deployment component to refer to the EJB
- `ejb-ref-type`: Type of the EJB (Entity/Session)
- `home`: Fully qualified name of the EJB's home interface
- `remote`: Fully qualified name of the EJB's remote interface
- `ejb-link`: Specifies that the EJB reference is linked to an enterprise bean
- `description`: A text description of the EJB reference

A sample `ejb-ref` element is shown here:

```
<ejb-ref>
  <description>Employee bean/description>
  <ejb-ref-name>EmployeeBean</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.foobar.employee.EmployeeHome</home>
  <remote>com.foobar.employee.Employee</remote>
</ejb-ref>
```

env-entry

The `env-entry` element declares environment parameters for a Web application. Each `env-entry` has the following child elements:

- `env-entry-name`: The JNDI name of the deployment component's environment entry. This name is relative to the `java:comp/env` context, and must be unique within a context.
- `env-entry-type`: The type for the environmental entry (for example, `java.lang.Integer`, `java.lang.String`).
- `env-entry-value`: The value of the deployment component's environment entry.

The following example shows a sample `env-entry`:

```
<!-- Environment entry examples -->
   <env-entry>
       <env-entry-name>maxExemptions</env-entry-name>
       <env-entry-type>java.lang.Integer</env-entry-type>
       <env-entry-value>15</env-entry-value>
   </env-entry>
   <env-entry>
       <env-entry-name>minExemptions</env-entry-name>
       <env-entry-type>java.lang.Integer</env-entry-type>
       <env-entry-value>1</env-entry-value>
   </env-entry>
```

error-page

The `error-page` element specifies the mapping between an error code or Java exception type and a Web resource. It contains the following child elements:

- `error-code`: The HTTP error code.
- `exception-type`: The fully qualified class name of the Java exception type.

  Either the `error-code` or the `exception-type` should be specified in an `error-page` element, but not both.
- `location`: The location of the resource (that is, the error Web page) that handles the error. The location is relative to the root of the Web application, and must have a leading slash (`/`).

A sample `error-page` is shown in the following code:

```
<error-page>
    <error-code>404</error-code>
    <location>/myApp/jsp/notFound.jsp</location>
</error-page>
<error-page>
    <error-code>500</error-code>
    <location>/myApp/jsp/SystemErr.jsp</location>
</error-page>
```

filter

The `filter` element declares a filter in the Web application. Filters are Java classes that implement a specified servlet interface (`javax.servlet.Filter`) and provide useful functionality to a Web application. As the name suggests, they are used to "filter" the request before the Web application sees it, or the response before the client sees it. This allows them to be used for applications such as the following:

- Logging
- Compression (images, data)
- Authentication
- Encryption
- Caching
- Transforming content (for example, XML content using XSLT)

The filter is mapped to either a servlet or a URL pattern in the `filter-mapping` element, using the `filter-name` value as a reference key.

The filter element consists of the following subelements:

- `filter-name`: The name of the filter. This must be unique in the Web application, and should not be empty. This name must match the `filter-name` `filter-mapping` element described in the next section.
- `filter-class`: The fully qualified Java class name of the filter.
- `init-param`: Initialization parameters for the filter specified as name-value pairs. These have the same structure as the `context-param` element described earlier, and consist of the `param-name`, `param-value`, and `description` subelements.
- `description`: A text description of the filter.
- `display-name`: A short, descriptive name that can be used by tools while displaying the filter configuration.
- `icon`: The `icon` element specifies icons that can be used by tools to symbolically represent the filter in GUI tools. It has two subelements: a `small-icon` and `large-icon` (see the `icon` element, described later in this chapter).

A sample `filter` configuration is shown in the following example:

```
<filter>
   <filter-name>Compression Filter</filter-name>
   <filter-class>compressionFilters.CompressionFilter</filter-class>
   <init-param>
     <param-name>compressionThreshold</param-name>
     <param-value>10</param-value>
   </init-param>
   <init-param>
     <param-name>debug</param-name>
     <param-value>0</param-value>
   </init-param>
</filter>
<filter>
   ...
</filter>
```

filter-mapping

As specified earlier, the filter is mapped to either a servlet or a URL pattern in the `filter-mapping` element, using the `filter-name` value for reference. The `Compression Filter` was declared in the `filter` element in the previous example, and the following example shows it being mapped to URL patterns that begin with `/CompressionTest`:

```
<filter-mapping>
    <filter-name>Compression Filter</filter-name>
    <url-pattern>/CompressionTest</url-pattern>
</filter-mapping>
```

The `filter-mapping` element can contain the following subelements:

- `filter-name`: The filter name. This *must* match the `filter-name` specified in the `filter` element.
- `url-pattern`: The URL pattern to which the filter applies.

New in Servlet 2.5: *The* `dispatcher` *element.*

- `dispatcher`: The dispatcher element indicates what kind of requests the filter is applied to. It can take the values `REQUEST`, `FORWARD`, `INCLUDE`, or `ERROR` (or a combination). The `REQUEST` value indicates that the filter is to be applied to requests that come directly from the client. The `FORWARD` value is for requests that come via a "forward" call. `INCLUDE` is for "included" Web pages. Finally, the `ERROR` filter applies if the request is being processed with the error page mechanism (see the `error-page` element).
- `servlet-name`: The servlet to which the filter applies. Servlet 2.4 allowed either `url-pattern` or `servlet-name`, but not both. In Servlet 2.5, both are allowed.

New in Servlet 2.5: *Improved filter mapping — wild cards and multiple mappings.*

Previously in Servlet 2.4 wildcards were not permitted in the filter mapping. Servlet 2.5 now allows wildcards, enabling you to do something like this:

```
<filter-mapping>
    <filter-name>Logging Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

In the previous example, the Logging filter is applied to all servlets and static content because the /* wildcard matches everything.

As mentioned earlier, Servlet 2.5 allows for multiple `servlet-name` and `url-pattern` mappings. The Servlet container handles this by internally expanding these mappings into multiple `filter-mapping` declarations. The example taken from the Servlet 2.5 specifications illustrates this:

```
<filter-mapping>
    <filter-name>Multiple Mappings Filter</filter-name>
    <url-pattern>/foo/*</url-pattern>
    <servlet-name>Servlet1</servlet-name>
    <servlet-name>Servlet2</servlet-name>
    <url-pattern>/bar/*</url-pattern>
</filter-mapping>
```

This is equivalent to specifying four mappings one after the other, as shown here:

```
<filter-mapping>
    <filter-name>Multiple Mappings Filter</filter-name>
    <url-pattern>/foo/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>Multiple Mappings Filter</filter-name>
    <servlet-name>Servlet1</servlet-name>
</filter-mapping>
<filter-mapping>
    <filter-name>Multiple Mappings Filter</filter-name>
    <servlet-name>Servlet2</servlet-name>
</filter-mapping>
<filter-mapping>
<filter-name>Multiple Mappings Filter</filter-name>
<url-pattern>/bar/*</url-pattern>
</filter-mapping>
```

icon

The `icon` element specifies icons that can be used by GUI tools to symbolically represent the parent element. It can occur under the `web-app` element (specifying icons to represent the Web application) or other elements (for example, the `filter` element described earlier). It has two subelements:

- `small-icon`
- `large-icon`

These set the small and large icon images, respectively. The images are relative path names to `gif` or `jpeg` files. The following is an example `icon` element:

```
<icon>
   <small-icon>an-icon16x16.jpg</small-icon>
   <large-icon>an-icon32x32.jpg</large-icon>
</icon>
```

jsp-config

The `jsp-config` element is used to configure JSP files in the Web application. It has the following child elements:

- `taglib`: Configures tag libraries used within the JSP pages. This is done via its two child elements: `taglib-uri` and the `taglib-location` (the location of the tag configuration `.tld` file).
- `jsp-property-group`: Configures JSP pages. This in turn has a number of child elements of its own:
  - `url-pattern`**:** The URL pattern for the JSPs.

> **If a URL pattern is also specified in the `servlet-mapping` (described later in this section), then the more specific pattern applies. If both match, then the `jsp-property-group` takes precedence.**

  - `el-ignored`: Sets the `isELIgnored` property for JSP pages. EL evaluation is enabled by default. JSP EL is a new expression language for accessing data from JSP pages.
  - `page-encoding`: The encoding to be used for the page (for example, ISO-8859-1).
  - `scripting-invalid`: Used to disable scripting in JSP pages (enabled by default).
  - `is-xml`: If `true`, implies that the documents matching the pattern are JSP pages, and can be interpreted as XML.
  - `include-prelude`: Specifies the path to a Web resource to be included in the beginning of the JSP page.
  - `include-coda`: Specifies the path to a Web resource to be included at the end of the JSP page.
  - `description`: A text description of the filter.
  - `display-name`: A short, descriptive name that can be used by tools while displaying the filter configuration.
- `icon`: The icon element specifies icons that can be used by GUI tools to symbolically represent the filter. It has two subelements: a `small-icon` and `large-icon` (see the `icon` element, described in more detail earlier in the chapter).

The `jsp-config` element for the example JSPs bundled along with Tomcat is shown here:

```
<jsp-config>
  ...
 <taglib>
     <taglib-uri>
        http://jakarta.apache.org/tomcat/examples-taglib
     </taglib-uri>
     <taglib-location>
        /WEB-INF/jsp/example-taglib.tld
     </taglib-location>
 </taglib>
 ...
 <jsp-property-group>
     <description>
         Special property group for JSP Configuration JSP example.
     </description>
     <display-name>JSPConfiguration</display-name>
     <url-pattern>/jsp2/misc/config.jsp</url-pattern>
     <el-ignored>true</el-ignored>
     <page-encoding>ISO-8859-1</page-encoding>
     <scripting-invalid>true</scripting-invalid>
     <include-prelude>/jsp2/misc/prelude.jspf</include-prelude>
     <include-coda>/jsp2/misc/coda.jspf</include-coda>
 </jsp-property-group>
</jsp-config>
```

listener

The `listener` element specifies the deployment properties for an application listener bean. It has the following subelements:

- `listener-class`: The fully qualified class name of the Java class corresponding to the listener.
- `description`: A text description of the listener.
- `display-name`: A short, descriptive name that can be used by tools while displaying the listener configuration.
- `icon`: The icon element specifies icons that can be used by GUI tools to symbolically represent the listener. It has two subelements: a `small-icon` and `large-icon` (see the `icon` element, described in more detail earlier in the chapter).

A sample `listener` element is shown here:

```
<listener>
    <listener-class>listeners.ContextListener</listener-class>
</listener>
<listener>
    <listener-class>listeners.SessionListener</listener-class>
</listener>
```

locale-encoding-mapping-list

This element contains the `locale-encoding-mapping` element that specifies the mapping between the locale and the encoding. The `locale-encoding-mapping` element has two child elements:

- `locale`: The locale to be encoded
- `encoding`: The encoding to be used

A sample is shown here:

```
<locale-encoding-mapping-list>
    <locale-encoding-mapping>
        <locale>en</locale>
        <encoding>en_US</encoding>
    </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

login-config

This element is used to configure the authentication method, the Realm name, and the attributes needed for `FORM`-based login. It has the following child elements:

- `auth-method`: The authentication method to be used. It must be one of the following: `BASIC`, `DIGEST`, `FORM`, or `CLIENT-CERT`.
- `realm-name`: The name of the Realm.
- `form-login-config`: If `FORM`-based authentication is used, this element is used to configure it. It specifies the form's login page (`form-login-page` element) and the error page (`form-error-page` element).

The `login-config` element is described in more detail in Chapter 14. A sample `login-config` is shown here:

```
<!-- Default login configuration uses form-based authentication -->
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>Example Form-Based Authentication Area</realm-name>
    <form-login-config>
       <form-login-page>/security/protected/login.jsp</form-login-page>
       <form-error-page>/security/protected/error.jsp</form-error-page>
    </form-login-config>
</login-config>
```

message-destination

The `message-destination` element specifies a message destination. The destination specified here is mapped to a physical destination by the deployer. It consists of the following child elements:

- `message-destination-name`: The name of the message destination. This name must be unique across all message destinations described in the deployment descriptor.

- `description`: A text description of the destination.
- `display-name`: A short, descriptive name that can be used by tools while displaying the destination.
- `icon`: The `icon` element specifies icons that can be used by GUI tools to symbolically represent the `message-destination`. It has two subelements: a `small-icon` and `large-icon` (see the `icon` element, described in more detail earlier in the chapter).

message-destination-ref

The `message-destination-ref` element declares a reference to a message destination associated with a resource in the deployment component's environment. It consists of the following child elements:

- `message-destination-ref-name`: Name of the message destination reference. This name is a JNDI name, relative to the `java:comp/env` context, and must be unique within the deployment descriptor.
- `message-destination-type`: Type of the destination. The type is specified as a fully qualified Java interface that is implemented by the destination.
- `message-destination-usage`: Specifies the use of the message destination. The destination is used for consuming messages (`Consumes`), producing messages (`Produces`), or both (`Both`).
- `message-destination-link`: Links the message destination reference to a message destination (see the `message-destination` element described earlier) or a message-driven bean. This value should match the `message-destination-name` defined in the `message-destination` element.
- `description`: Used for documentation.

mime-mapping

The `mime-mapping` element specifies the mapping between the extension for a resource and its MIME type. It has two child elements for this: `extension` and `mime-type`. A sample `mime-mapping` is shown in the following example:

```
<mime-mapping>
    <extension>pdf</extension>
    <mime-type>application/pdf</mime-type>
</mime-mapping>
```

resource-env-ref

This element contains a reference to the administered object associated with a resource. It has the following child elements:

- `resource-env-ref-name`: Name of the resource environment reference. This name is a JNDI name, relative to the `java:comp/env` context, and must be unique within the deployment descriptor.
- `resource-env-ref-type`: The type of the resource environment reference. This must be the fully qualified name of a Java class or interface.
- `description`: Used for documentation.

resource-ref

The `resource-ref` element specifies a reference to an external resource. It consists of the following child elements:

- `res-ref-name`: Name of the resource manager connection factory reference. This name is a JNDI name, relative to the `java:comp/env` context, and must be unique within the deployment descriptor.
- `res-type`: Type of the data source. The type is specified as a fully qualified Java class or interface that is implemented by the data source.
- `res-auth`: Specifies whether the deployment component code signs on programmatically to the resource manager (`Application`), or whether the container signs on to the resource manager on its behalf (`Container`). If the container handles this, the deployer needs to supply information for the sign-on.
- `res-sharing-scope`: Specifies whether the connections obtained through the resource manager are sharable (`Sharable`) or not (`Unsharable`).
- `description`: Used for documentation.

The example `resource-ref` element shown here is a reference to a JDBC DataSource:

```
<!-- JDBC DataSources (java:comp/env/jdbc) -->
  <resource-ref>
      <description>The default JDBC datasource</description>
      <res-ref-name>jdbc/DefaultDS</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
  </resource-ref>
```

security-constraint

This element specifies the security constraints on one or more Web resource collections, as follows:

- `display-name`: The display name for the security constraint.
- `web-resource-collection`: Specifies the resources (`url-pattern` element) and the HTTP methods (`http-method` element) that are allowed on these resources.
- `auth-constraint`: Indicates the user roles (`role-name` element) that are permitted to access the Web resources protected by this security constraint. These role names must match those defined in the `security-role` element described later in the chapter. The pattern `*` matches all roles defined in the Web application.
- `user-data-constraint`: Specifies how the data transmitted between the client and the Servlet container is protected. This is done via its `transport-guarantee` child element, and it can be set to `NONE`, `INTEGRAL`, or `CONFIDENTIAL`.

A sample `security-constraint` is shown in the following example:

```
<security-constraint>
      <display-name>Example Security Constraint</display-name>
      <web-resource-collection>
           <web-resource-name>Protected Area</web-resource-name>
               <!-- Define the context-relative URL(s) to be protected -->
           <url-pattern>/security/protected/*</url-pattern>
               <!-- If you list http methods, only those methods are protected -->
               <http-method>DELETE</http-method>
           <http-method>GET</http-method>
           <http-method>POST</http-method>
               <http-method>PUT</http-method>
      </web-resource-collection>
      <auth-constraint>
           <!-- Anyone with one of the listed roles may access this area -->
           <role-name>tomcat</role-name>
               <role-name>role1</role-name>
      </auth-constraint>
</security-constraint>
```

security-role

The `security-role` element lists all the security roles used in the Web application. These role names are specified via the `role-name` child element, and are used in the `security-constraint` (see the previous element) to specify the security constraints for a Web application. A sample `security-role` element is shown in the following example that corresponds to the role used in the `security-constraint` in the previous example:

```
<security-role>
    <role-name>role1</role-name>
</security-role>
<security-role>
    <role-name>tomcat</role-name>
</security-role>
```

service-ref

A `service-ref` element declares a reference to a Web service. It consists of the following child elements:

- `service-ref-name`: The logical name that components in the module use to look up the service. We recommend that this name start with `/service/`.
- `service-interface`: The fully qualified class name of the JAX-RPC Service interface on which the client depends.
- `wsdl-file`: The URI location for the WSDL file. The location is relative to the Web application root.
- `jaxrpc-mapping-file`: File that specifies the JAX-RPC mapping between the Java interfaces used by the application and the descriptions in the WSDL file.
- `service-qname`: The name of the WSDL service element.

- `port-component-reference`: This element declares the service endpoint interface or provides the link to a port component that specifies this. It has two child elements:
  - `service-endpoint-interface`: A fully qualified Java class that represents the service endpoint interface of a WSDL port
  - `port-component-link`: Links the port component reference to a specific port component
- `handler`: Declares the handler for the port component. This, in turn, has a number of child elements:
  - `handler-name`: Name of the handler. The name must be unique within the module.
  - `handler-class`: Fully qualified Java class name for the handler.
  - `init-param`: This contains parameter name (`param-name`) and value (`param-value`) pairs for initialization parameters.
  - `soap-header`: Qualified name (`qName`) of the SOAP header that will be processed by this handler.
  - `soap-role`: SOAP actor definitions that the handler will play as a role.
  - `port-name`: WSDL port name that the handler is associated with.
- `description`: A text description of the service reference.
- `display-name`: A short, descriptive name that can be used by tools while displaying the service reference.
- `icon`: The `icon` element specifies icons that can be used by GUI tools to symbolically represent the element. It has two subelements: a `small-icon` and a `large-icon` (see the `icon` element, described in more detail earlier in the chapter).

servlet

The `servlet` element is used to configure a servlet or JSP file. It consists of the following child elements:

- `servlet-name`: The name of the servlet. This must be unique across the Web application.
- `servlet-class`: The fully qualified Java class name of the servlet.
- `jsp-file`: The full path of the JSP file within the Web application (that is, beginning from `/`). If the `load-on-startup` element is enabled (described later), then the JSP should be precompiled.

> **Only one of the `servlet-class` or `jsp-file` elements should be specified.**

- `init-param`: The `init-param` element is used to pass initialization time parameters to the servlet. This is done via its `param-name` and `param-value` elements. It also has a `description` element that is used to document the parameters.
- `load-on-startup`: The `load-on-startup` element indicates to the Servlet container that this servlet should be loaded at startup time. This element can also contain an optional positive integer value that specifies the startup sequence. (Lower-integer–valued servlets are loaded before

the higher-integer–valued ones.) A negative or missing value indicates that the order doesn't matter.

- `run-as`: The security role to be used for the execution of the servlet or JSP page. This has two child elements: an optional `description` and a `role-name` that specifies the role.
- `security-role-ref`: This element declares the security role reference in a component's code. It consists of the security role name (`role-name` element) and a link to the security role (`role-link` element). It also has an optional `description` element, again used for documentation purposes.
- `description`: A text description of the listener.
- `display-name`: A short, descriptive name that can be used by tools while displaying the listener configuration.
- `icon`: The `icon` element specifies icons that can be used by GUI tools to symbolically represent the listener. It has two subelements: a `small-icon` and `large-icon` (see the `icon` element, described in more detail earlier in the chapter).

A sample `servlet` element is shown here:

```
<servlet>
    <servlet-name>org.apache.jsp.num.numguess_jsp</servlet-name>
    <servlet-class>org.apache.jsp.num.numguess_jsp</servlet-class>
</servlet>
```

servlet-mapping

The `servlet-mapping` element defines the mapping between a servlet (`servlet-name` element) and a URL pattern (`url-pattern` element). The `servlet-name` must match the name defined in the `servlet` element, as shown in the following example:

```
<servlet-mapping>
    <servlet-name>org.apache.jsp.num.numguess_jsp</servlet-name>
    <url-pattern>/num/numguess.jsp</url-pattern>
</servlet-mapping>
```

New in Servlet 2.5: *Wildcards and multiple matching in* `url-pattern`.

Just like `filter-mapping`, `servlet-mapping` now allows wildcards and multiple mappings, as shown in this example:

```
<servlet-mapping>
    <servlet-name>color</servlet-name>
    <url-pattern>/color/*</url-pattern>
    <url-pattern>/colour/*</url-pattern>
</servlet-mapping>
```

session-config

This element defines the session parameters for the Web application. It has a `session-timeout` element, which specifies the default session timeout interval, in minutes, for all sessions for this application.

The following example specifies a session timeout of 30 minutes:

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

If set to 0 (zero) or less, the session is set to never timeout.

welcome-file-list

This element contains an ordered list of welcome files (for example, `index.html`), and is specified via the `welcome-file` child element. This file is displayed when someone browses to the Web application URL:

```
http://hostname:port/<web application name>/
```

Following is a sample `welcome-file-list` element:

```
<welcome-file-list>
    <welcome-file>index.jsp<welcome-file>
    <welcome-file>index.html<welcome-file>
    <welcome-file>home.html<welcome-file>
</welcome-file-list>
```

If more than one of these files are present in the Web application, then the order in which they are specified determines which one is shown — the file listed earlier has higher precedence. If none of the files in the example list are found, then, depending on the configuration, an HTTP 404–Not Found error message is displayed.

Summary

Configuring Web applications on production and test systems is an important part of an administrator's job. This involves tasks such as adding or removing filters for given URL patterns, session configuration, error page configuration, the addition of tag libraries, and the configuration of initialization parameters for the Web application. Understanding the Web application structure and the deployment descriptor is, therefore, important for administrators. This chapter focused on these issues, and described the various elements that make up a Web application. It highlighted the following aspects:

- The directory structure for a typical Web application
- A detailed examination of both the older Servlet 2.3–style (DTD-based) deployment descriptor and the Servlet 2.4/2.5–style (schema-based) deployment descriptor.

Chapter 8 covers how Web applications are managed within Tomcat.

Only one of the `servlet-class` or `jsp-file` elements should be specified.

8

Web Application Administration

This chapter discusses how to manage Web applications using the tools provided by Tomcat. Tomcat's management tools include the manager application and the new virtual-host manager. The manager application helps manage Web applications. It enables administrators to deploy Web applications, view deployed applications, and undeploy Web applications when necessary. While these tasks can also be performed manually by editing Tomcat's configuration files, this method requires Tomcat to be restarted. The manager application automates these tasks and enables them to be performed on a running instance of Tomcat. This way, other Web applications that are already running in the same Tomcat container are left undisturbed.

This chapter covers the `manager` application. The chapter first looks at a sample Web application that is used as an example throughout the chapter. Then it discusses in detail the `manager` application, including the following areas:

- Enabling access to the manager application
- The three ways of interacting with the manager application (the Web-based user interface, Ant scripts, and HTTP commands)
- Security conlsiderations while using the manager application

Sample Web Application

This chapter uses a simple Web application for testing the `manager` commands. This application consists of nothing more than one HTML file and one JSP file.

The HTML file (`index.html`) contains a form that asks for the user's name and uses HTTP `POST` to send the result to a JSP page:

```
<html>
  <head>
    <title>Hello Web Application</title>
  </head>

  <body>
    <h1>Hello Web Application</h1>
    <form action="/hello/hello.jsp" method="POST" >
      <table width="75%">
        <tr>
          <td width="48%">What is your name?</td>
          <td width="52%">
            <input type="text" name="name" />
          </td>
        </tr>
      </table>
      <p>
        <input type="submit" name="Submit" value="Submit name" />
        <input type="reset" name="Reset" value="Reset form" />
      </p>
    </form>
  </body>
</html>
```

The JSP page (`hello.jsp`) then prints a `Hello <name>` message. The `<name>` portion is the name that the user entered in the `index.html` form:

```
<html>
  <head>
    <title>Hello Web Application</title>
  </head>
  <body>
    <h1>Hello Web Application</h1>
    <%
      String name  = request.getParameter("name");
      if ((name == null) || (name.trim().length() == 0)) {
    %>
        You didn't tell me your name!
    <%
      } else {
    %>
        Hello <%=name%>.
    <%
      }
    %>
    <a href="/hello/index.html">Try again?</a>
  </body>
</html>
```

This Web application will be deployed with the `/hello` context path; therefore, `http://localhost:8080/hello/index.html` would be the URL to access it.

The commands for building the WAR file are as follows:

```
$ cd /path/to/hello
$ jar cvf hello.war .
```

There is also a sample Ant build script for building this WAR file, which is discussed later in the chapter and included with the code download for the book at `wrox.com`. The `/path/to/hello` is the directory in which the `index.html` and `hello.jsp` files reside.

How do you get a Web application deployed to the default content, i.e., so that you can access it as `http://localhost:8080/index.html`*?*

One way to do this is to undeploy the default Web application, name your WAR file `ROOT.war` *and deploy it. This WAR file overwrites the default Web application, deployed at* `<TOMCAT_HOME>/webapps/ROOT`*. Deployment and undeployment of Web applications are covered in the next few sections.*

Tomcat Manager Application

As mentioned earlier, the Tomcat `manager` application is a Web application that enables you to carry out various system administration tasks related to deploying, undeploying, and managing a Web application.

The following are the three ways to interact with the `manager` application:

- **Using the Web interface to the Admin application:** This is discussed in the section "Tomcat Manager: Web Interface."
- **Using the Ant-based interface:** This is covered in the section "Tomcat Manager: Managing Applications with Ant," later in this chapter.
- **Using HTTP requests:** This can be done either via the browser or by writing scripts to automate the process. The section "Tomcat Manager: Using HTTP Requests" later in this chapter covers this in more detail.

The first two mechanisms are the more commonly used — and more convenient — ways to manage Web applications. The HTTP request mechanism is an older mechanism that is primarily used by automated scripts, although even here, the Ant-based interface is more convenient. Future versions of Tomcat might add a Web service interface, thus enabling easier integration of the Tomcat management tasks from third-party applications.

Access to the manager application is restricted to authorized users. This prevents unauthorized users from undeploying (or deploying) applications, or performing any other operation that they shouldn't. The next section discusses how this access control is configured and then examines the other configuration parameters for the manager application. Finally, it describes all the manager application commands in more detail.

Following is a summary of some of the tasks that the manager application can perform:

- Deploy a new Web application
- List the currently deployed Web applications, as well as the sessions that are currently active for those Web applications
- Reload an existing Web application
- List the available global JNDI resources
- List the available security roles
- Start a stopped application
- Stop an existing application, but not undeploy it
- Undeploy a Web application
- Display session statistics

An application can be deployed manually, too. Following are the ways to do this:

- Add a `<Context>` entry in Tomcat's `server.xml` configuration file. This enables you to place the Web application in a location other than the default `$CATALINA_HOME/webapps` directory.
- Copy the entire application directory into the `$CATALINA_HOME/webapps` directory. The `server.xml` file does not have to be edited in this case.
- Copy the WAR file for the application into the `$CATALINA_HOME/webapps` directory. In this option, too, the `server.xml` file does not have to be edited.

However, there are advantages to using a `manager` application. First, all these methods of deploying just described require you to restart Tomcat. When deploying is done via the `manager` application, Tomcat is not restarted and, hence, the other running Web applications are not affected.

Alternatively, the `autoDeploy` attribute in the `Host` (see `server.xml`) could be set to `true`, which is actually the default in Tomcat 6. This causes any Web application dropped into Tomcat's application base to be deployed automatically. Doing this compromises performance, however, because Tomcat must continually monitor the application base directory; hence, `autoDeploy` is often set to `false`. If `autoDeploy` is set to `false`, then copying the Web application or the WAR file to the `webapps` directory will not deploy the Web application. You would need to either restart Tomcat or use the `deploy` command.

Another advantage of using the `manager` application is that it supports remote installs. That is, the Web application directory (or WAR file) doesn't need to be transferred via FTP or some other means to the host machine running Tomcat. The `deploy` command takes care of transferring the Web application WAR file from the local development machine to the remote machine running the Tomcat server.

Enabling Access to the Manager Application

Before using the manager application, the server needs to be configured to enable access. Access to this application is controlled via a Security Realm. Any Realm implementation can be used (Memory, User Database, JDBC, JNDI, or JAAS). This example uses the default Realm, the User Database Realm, for simplicity.

In a User Database Realm, the usernames and their supporting information are stored in memory and are initialized at startup from an XML configuration file (`$CATALINA_HOME/conf/tomcat-users.xml`) kept on the file system. This file needs to be edited to add a user with a role of `manager`. In the following entry, the username and password for this role are `admin` and `secret`, respectively:

```
<tomcat-users>
  ...
  <user username="admin" password="secret" roles="manager" />
  ...
</tomcat-users>
```

Tomcat now needs to be restarted to make it reread the `tomcat-users.xml` file. To determine whether the `manager` application setup was successful, browse to the default Tomcat URL (`http://localhost:8080`), and click the Tomcat Manager link, or go directly to the manager application URL `http://localhost:8080/manager/html`. The user is then prompted for a username and password. After entering the values set in the `tomcat-users.xml` file, the Web page of the manager application shown in Figure 8-1 should be displayed.

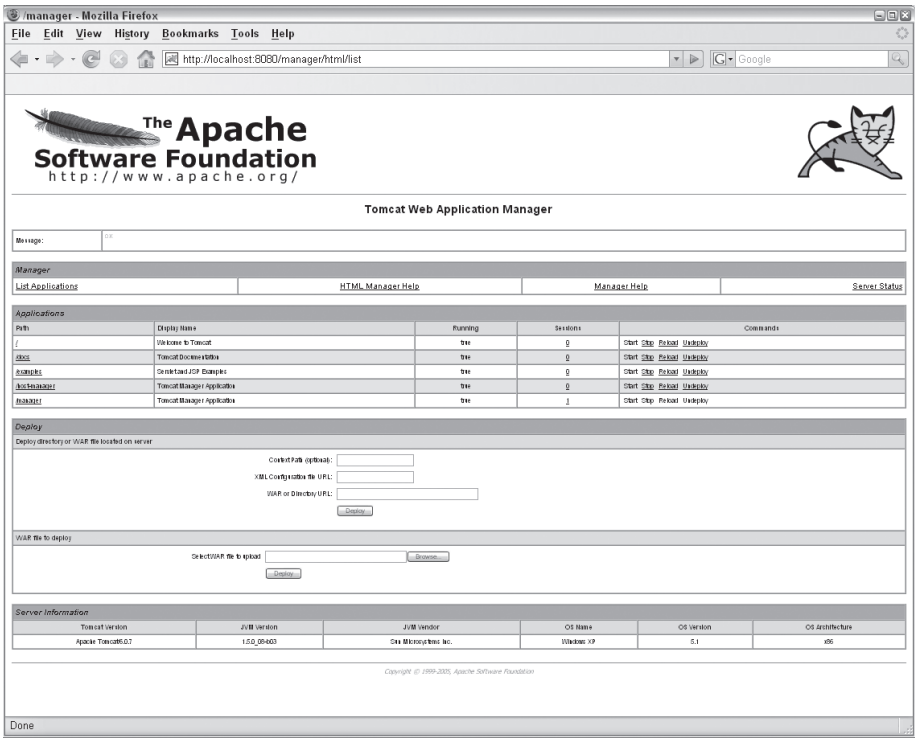

Figure 8-1: The Tomcat Web Application Manager

Keeping the password for the manager application in clear text may be okay for a development environment, but not suitable for anything else. The "Security Considerations" section at the end of the chapter addresses this and other security concerns.

As you can see in the figure, the response for a successful command execution (the `Message` parameter) is an `OK` string. A missing `OK` is an indication of failure, and the rest of the message provides the cause. The possible causes of failure for each command are listed later in this chapter in the section "Possible Errors."

> *During installation of Tomcat on Windows, using the installer executable (and not the Zip file), the installer asks the user for the admin username and password. The username and password entered at install time are used to generate entries for the* `tomcat-users.xml` *file for both the manager and admin Web applications. Hence, no configuration is needed in this case, unless another user with manager privileges is to be added or the manager password is to be changed.*

Manager Application Configuration

The previous section looked at `tomcat-users.xml`, which defines the username and password for the manager role. The other manager application–related configuration parameters are the *manager context entry* and the *deployment descriptor*.

No changes have to be made for the manager application to work — the settings are configured by default. They can, however, be modified for deployment requirements — for example, to change the security constraints for the manager application, to change the authentication mechanism for users in the manager role, or even to change the name of the role from "manager" to some other name if required. This section covers these configurable parameters for the manager application.

Manager Application Context Entry

In Tomcat 6, unlike Tomcat 5/5.5, the `manager` context is configured in the same way as the other application contexts.

Following is the default configuration for the manager application context from the `$CATALINA_HOME/webapps/manager/META-INF/context.xml` file:

```
<Context antiResourceLocking="false" privileged="true" />
```

This only difference from a standard web application is the setting of the privileged attribute to `true`. This enables the application to access the container's servlets. This attribute is `false` for a normal Web application deployed in Tomcat.

In Tomcat 5/5.5, the context was defined in the `$CATALINA_HOME/server/webapps/manager/manager.xml` file.

Manager Application Deployment Descriptor

The `tomcat-users.xml` file shown earlier defined the username and password for the `manager` role. This section discusses how the security constraints for this role are specified. The manager application's deployment descriptor for versions prior to Tomcat 4.1 is `$CATALINA_HOME/webapps/manager/WEB-INF/web.xml`. In Tomcat 5, the `web.xml` file moved to `$CATALINA_HOME/server/webapps/manager/WEB-INF`; however, it is back to the original location under `$CATALINA_HOME/webapps/manager/WEB-INF/` in Tomcat 6.

The `web.xml` (the deployment descriptor) defines, among other things, the security constraints on the manager application. The following snippet describes the default security constraint definition for the `manager` Web application. The `<role-name>` defined here (shown in bold) specifies that only users in that role can access the manager Web application:

```
<!-- Define a Security Constraint on this Application -->
<security-constraint>
    <web-resource-collection>
      <web-resource-name>HTMLManger and Manager command</web-resource-name>
      <url-pattern>/jmxproxy/*</url-pattern>
      <url-pattern>/html/*</url-pattern>
      <url-pattern>/list</url-pattern>
      <url-pattern>/sessions</url-pattern>
      <url-pattern>/start</url-pattern>
      <url-pattern>/stop</url-pattern>
      <url-pattern>/install</url-pattern>
      <url-pattern>/remove</url-pattern>
      <url-pattern>/deploy</url-pattern>
      <url-pattern>/undeploy</url-pattern>
      <url-pattern>/reload</url-pattern>
      <url-pattern>/save</url-pattern>
      <url-pattern>/serverinfo</url-pattern>
      <url-pattern>/status/*</url-pattern>
      <url-pattern>/roles</url-pattern>
      <url-pattern>/resources</url-pattern>
    </web-resource-collection>
    <auth-constraint>
       <!-- NOTE:  This role is not present in the default users file -->
      <role-name>manager</role-name>
     </auth-constraint>
</security-constraint>
```

The authentication mechanism for the `manager` application is also defined here. The default setting is `BASIC` authentication, as shown in the following code. Administrators could set up a more rigorous mechanism for manager application authentication — for example, a client-certificate–based mechanism (`<auth-method>` set to `CLIENT-CERT`):

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Tomcat Manager Application</realm-name>
</login-config>
```

The `<security-role>` lists all the roles that can log in to the manager application. In this case, it is restricted to only one user role (that is, the manager role):

```
<!-- Security roles referenced by this web application -->
<security-role>
  <description>
    The role that is required to log in to the Manager Application
  </description>
  <role-name>manager</role-name>
</security-role>
```

Tomcat Manager: Web Interface

Tomcat has a Web interface for the manager application that enables you to start, stop, remove, reload, and deploy new Web applications without having to modify configuration files.

To access the Tomcat manager Web application, access `http://localhost:8080/manager/html` (or the actual hostname/port, as the case may be). You will then be prompted for a username and password. Use the same username and password that you entered into `tomcat-users.xml`. This will take you to the `manager` application home page, as shown in Figure 8-1.

Displaying Tomcat Server Status

The `status` command retrieves miscellaneous information about a running instance of Tomcat (such as the free and total memory used by the JVM, number of threads, and so on). This is accessible via the `Server Status` link. Figure 8-2 shows the output of this command.

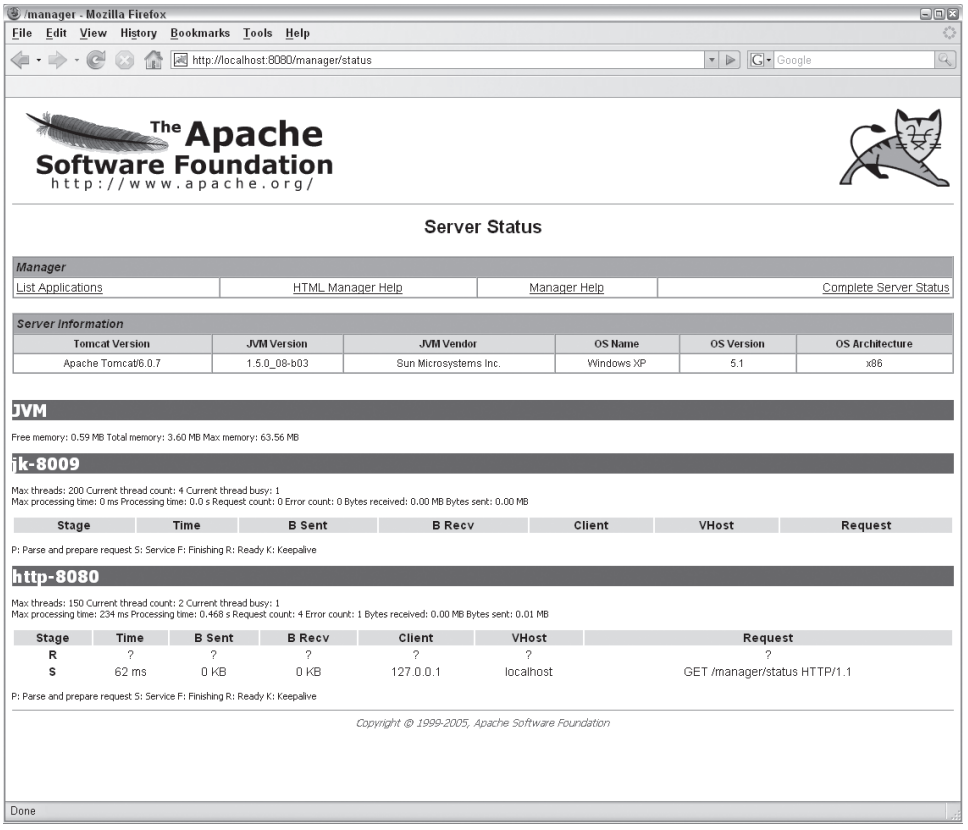

Figure 8-2: Output of the server status command

Information such as the class load time, processing time, and so on, on every servlet class/JSP for all the deployed Web applications, can be obtained using the Complete Server Status option (visible while viewing the server status).

Managing Web Applications

Administrators can start, stop, reload, and remove Web applications by clicking on the relevant links provided at the end of each application (see Figure 8-3).

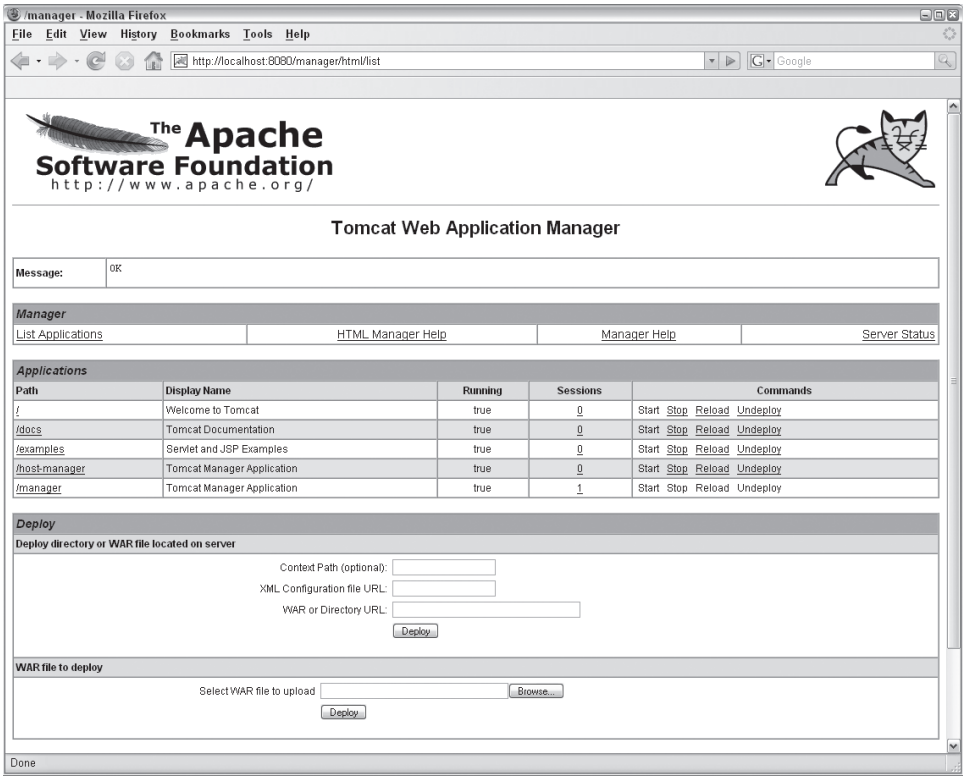
/manager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/manager/html/list
Google
The Apache Software Foundation
http://www.apache.org/
Tomcat Web Application Manager
Message: OK
Manager
List Applications HTML Manager Help Manager Help Server Status
Applications
Path Display Name Running Sessions Commands
/ Welcome to Tomcat true 0 Start Stop Reload Undeploy
/docs Tomcat Documentation true 0 Start Stop Reload Undeploy
/examples Servlet and JSP Examples true 0 Start Stop Reload Undeploy
/host-manager Tomcat Manager Application true 0 Start Stop Reload Undeploy
/manager Tomcat Manager Application true 1 Start Stop Reload Undeploy
Deploy
Deploy directory or WAR file located on server
Context Path (optional):
XML Configuration file URL:
WAR or Directory URL:
Deploy
WAR file to deploy
Select WAR file to upload
Browse...
Deploy
Done

Figure 8-3: Commands for Tomcat Manager's Web interface

Click the List Application link. You should see the Web page shown in Figure 8-3. The Applications table has five columns:

- **Path:** Lists the Web application path. The pathname links to the URL for the Web application.
- **Display Name:** Picked up from the `<display-name>` element in the application's deployment descriptor (`web.xml`).
- **Running:** Indicates the running status for the application (`true` if the application is running, and `false` otherwise).

- **Sessions:** Indicates the number of active sessions for the Web application. Clicking the link for the number of sessions returns the session statistics for that particular Web application.
- **Commands:** These are the links to the `start`, `stop`, `reload`, and `remove` commands for the Web application. The manual commands for this were shown earlier, but using the Web application manager saves the effort (and possible errors) of typing a command URL for performing these tasks.

The commands are pretty much self explanatory: the List Applications link shows all deployed Web applications, and each `Start/Stop/Reload/Undeploy` link allows you to perform the desired task.

- **Deploy:** The WAR file for the Web application is uploaded from the client machine to the machine on which Tomcat is running, and copied into the application base directory of the given virtual host. The deployed application is automatically started (i.e., available for use by Web clients).
- **Stop:** Send a signal to the Web application to stop. The application is then no longer available to users, although it still remains deployed. If the list command is run again, the state of the application is shown as "stopped."
- **Start:** Start up a stopped Web application.
- **Reload:** Signal the Web application to stop and then restart. This causes the class files and properties to be reread. The deployment descriptor is, however, not re-read.
- **Undeploy:** Signals the application to shut down (if it is still running) and then deletes the Web application directory and the application WAR file. If there is a `<Context>` entry for the Web application in `$CATALINA_HOME/conf/server.xml`, it gets deleted too.

Deploying a Web Application

A new Web application can be deployed using the manager application both locally ("Deploy directory or WAR file located on server") as well as remotely ("Upload a WAR file to deploy").

> *Some of the options available via the HTTP command interface (such as the* `tag`, `update`, *and* `pause` *parameters) are not currently usable from the Web interface.*

Figure 8-4 shows the manager Web interface after successfully executing the `deploy` command to deploy the `hello` application. As you can see, Tomcat adds another row for the application in the list of deployed applications.

Tomcat Manager: Managing Applications with Ant

The Tomcat management commands can also be run from an Ant build script. Ant is a Java-based build tool from the Apache Software Foundation. The capability to manage the Web application right from the build script is a very powerful feature. Appendix B provides a detailed introduction to Ant for those unfamiliar with it.

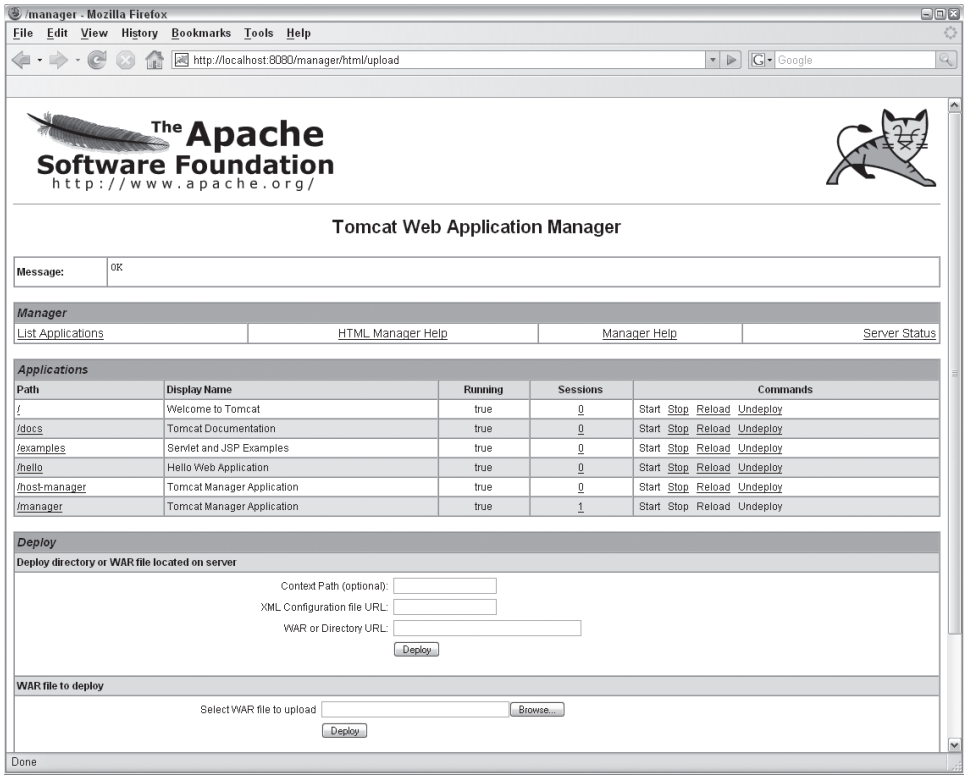
/manager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/manager/html/upload
The Apache Software Foundation
http://www.apache.org/
Tomcat Web Application Manager
Message: OK
Manager
List Applications HTML Manager Help Manager Help Server Status
Applications
Path Display Name Running Sessions Commands
/ Welcome to Tomcat true 0 Start Stop Reload Undeploy
/docs Tomcat Documentation true 0 Start Stop Reload Undeploy
/examples Servlet and JSP Examples true 0 Start Stop Reload Undeploy
/hello Hello Web Application true 0 Start Stop Reload Undeploy
/host-manager Tomcat Manager Application true 0 Start Stop Reload Undeploy
/manager Tomcat Manager Application true 1 Start Stop Reload Undeploy
Deploy
Deploy directory or WAR file located on server
Context Path (optional):
XML Configuration file URL:
WAR or Directory URL:
Deploy
WAR file to deploy
Select WAR file to upload
Browse...
Deploy
Done

Figure 8-4: Successful deployment of a Web application

Before you can start using Ant to manage Web applications, you need to do the following:

1. Install Ant if you do not already have it installed on your system. Refer to Appendix B for more details.
2. Copy the `$CATALINA_HOME/ lib/catalina-ant.jar` file into Ant's library directory (`$ANT_HOME/lib`) or add it to the Ant `CLASSPATH`. This JAR file contains the Tomcat management task definitions for Ant.
3. Add `$ANT_HOME/bin` to your `PATH`.
4. Add to your custom `build.xml` script the `<taskdef>` elements that call the Tomcat `manager` commands.
5. Of course, as with the Web-based manager UI, you need to add a user with the `manager` role to Tomcat's user database if such a user does not exist.

The following sample `build.xml` file builds and deploys the `hello` Web application discussed at the beginning of the chapter:

```
<project name="HelloApplication" default="dist" basedir=".">
    <property name="tomcat.home"  value="/path/to/tomcat"/>
    <property name="app.name"     value="hello"/>
```

(continued)

```
        <property name="context.path" value="/${app.name}"/>
        <property name="web.home"     value="${basedir}/web"/>
        <property name="dist.home"    value="${basedir}/dist"/>
        <property name="war.file"     value="${dist.home}/${app.name}.war"/>
```

The `<project>` tag has attributes for the name of the project and the default target. The default target in this case is called `dist` (i.e., create the distributable jar file). Running Ant with no options will invoke the tasks associated with this default target. The `basedir` attribute is the base directory for all path calculations in the Ant build script. This is set to "." (i.e., the current directory), so all the paths for the build process are assumed to be relative to the directory from which Ant is run. The properties for the build are defined next, such as the location of the target directory where the WAR file is generated. Typical Web applications have Java code in them (Java beans, servlets), and thus the build file would have a property pointing to the source directory. Having these properties is not mandatory, but merely a convenience to avoid putting the full path names in multiple places in the build script, and thus introducing potential sources of errors.

The following properties specify the access URL and username/password for the manager application. This password can be passed via the command line, or included from a property file.

```
<!-- Configure properties to access the Manager application -->
<property name="url"      value="http://localhost:8080/manager"/>
<property name="username" value="myusername"/>
<property name="password" value="mypassword"/>
```

The task definitions for the `manager` application now need to be specified. Ant allows for custom tasks that extend its functionality. Tomcat implements the custom tasks shown in the following example for executing the `manager` application commands. For example, `org.apache.catalina.ant.DeployTask` executes the `deploy` command against the manager application:

```
   <!-- Configure the custom Ant tasks for the Manager application -->
<taskdef name="deploy"
           classname="org.apache.catalina.ant.DeployTask"/>
   ...
<taskdef name="undeploy"
           classname="org.apache.catalina.ant.UndeployTask"/>
```

Instead of adding one line for each task, a more convenient mechanism is to list all tasks in a property file, as shown here:

```
deploy=org.apache.catalina.ant.DeployTask
list=org.apache.catalina.ant.ListTask
status=org.apache.catalina.ant.JKStatusUpdateTask
jmxget=org.apache.catalina.ant.JMXGetTask
jmxquery=org.apache.catalina.ant.JMXQueryTask
jmxset=org.apache.catalina.ant.JMXSetTask
reload=org.apache.catalina.ant.ReloadTask
remove=org.apache.catalina.ant.RemoveTask
resources=org.apache.catalina.ant.ResourcesTask
roles=org.apache.catalina.ant.RolesTask
serverinfo=org.apache.catalina.ant.ServerinfoTask
sessions=org.apache.catalina.ant.SessionsTask
start=org.apache.catalina.ant.StartTask
stop=org.apache.catalina.ant.StopTask
undeploy=org.apache.catalina.ant.UndeployTask
```

In this example, the previous property file is named `tomcat-tasks.property`. This then can be referenced in the build script as:

```
<!-- Classpath for Tomcat ant tasks -->
<path id="deployer.classpath">
    <fileset dir="${tomcat.home}/ lib">
        <include name="*.jar"/>
    </fileset>
</path>

<!-- Configure the custom Ant tasks for the Manager application -->
<taskdef file="tomcat-tasks.properties"
        classpathref="deployer.classpath" />
```

The `deployer.classpath` set in the path element in the previous code points to the JAR files needed by Ant for the Tomcat tasks, plus other JAR files as needed for building the Web application.

Beginning with Tomcat 5.5, there is a neat shortcut to create the `tomcat-tasks.property` file and set the JAR files for the Ant tasks. All you need to do is to add this fragment to your build script (assuming `tomcat.home` points to the right location):

```
<import file="${tomcat.home}/bin/catalina-tasks.xml"/>
```

The `catalina-tasks.xml` file ships with Tomcat 5.5 and 6, and sets everything (`taskdefs`, JAR files) required to run Tomcat's Ant tasks.

Next is the Ant target that performs initializations (in this case, creates the directories needed for the build):

```
<target name="init">
         <mkdir dir="${dist.home}"/>
</target>
```

The `compile` target is shown here. This has the Ant instructions to compile all the Java files into class files. The `hello` application doesn't have any class files, so nothing will be done, but any serious Web application will contain Java files. Notice how the `compile` task depends on the `init` task. This ensures that the initialization steps are performed before Ant compiles the Java files:

```
<target name="compile" description="Compile web application"
          depends="init">
</target>
```

The `dist` target builds the application WAR file:

```
<target name="dist" depends="compile"
             description="Creates the deployable WAR file">
         <war destfile="${war.file}"
              webxml="${web.home}/WEB-INF/web.xml">
             <fileset dir="${web.home}" excludes="**/web.xml" />
         </war>
</target
```

Finally, the manager tasks for listing all Web applications, and installing/uninstalling and deploying/undeploying Web applications are as follows:

```
<target name="deploy" depends="dist" description="Deploy web application">
         <deploy url="${tomcat.manager.url}" username="${tomcat.username}"
                 password="${tomcat.password}" path="${context.path}"
                 war="${war.file}" update="true" />
</target>

<target name="reload" description="Reload web application">
         <reload url="${tomcat.manager.url}" username="${tomcat.username}"
                 password="${tomcat.password}" path="${context.path}"/>
</target>

<target name="start" description="Start web application">
         <start url="${tomcat.manager.url}"
                username="${tomcat.username}" password="${tomcat.password}"
                path="${context.path}"/>
</target>

<target name="stop" description="Stop web application">
         <stop url="${tomcat.manager.url}" username="${tomcat.username}"
               password="${tomcat.password}" path="${context.path}"/>
</target>

<target name="undeploy" description="Undeploy web application">
         <undeploy url="${tomcat.manager.url}" username="${tomcat.username}"
                   password="${tomcat.password}" path="${context.path}"/>
</target>
```

Before using the Ant script, `$CATALINA_HOME/ lib/catalina-ant.jar` must be added to the `CLASSPATH`, and the Ant install directory must be added to the system path (Ant version 1.7.0 was used for this example):

```
$ CLASSPATH=$CLASSPATH:$CATALINA_HOME/ lib/catalina-ant.jar
$ PATH=$PATH:/path/to/ant1.7.0/bin
$ export CLASSPATH PATH
```

The `password` property in the Ant script contains the password for the user with manager privileges. This is useful for development environments in which developers don't want to specify the password each time. The password value can be overridden from the command line, or even omitted from the build file altogether and passed only from the command line. This avoids the security risk of putting the password in a clear text file:

```
$ ant --Dpassword=secret list
```

The capability to run the manager commands from within Ant files allows for a very integrated develop-deploy-test cycle for Web application development. For example, after developing the HTML pages, servlets, JSP pages, and other Java classes for the Web application, the developer would need to compile all the Java code:

```
$ ant compile
```

The `compile` target in `build.xml` compiles all the Java code and puts the class files into the appropriate location (such as the `/WEB-INF/classes` directory). It then builds the deployable JAR file. Developers may need to fix compilation errors, if any, and then rerun the `Ant` command. In our simple Web application, this task does nothing as there are no Java files to compile.

The `deploy` target can then be used to deploy the Web application in the Tomcat instance specified in the Ant build file:

```
$ ant deploy
```

This installed application can then be tested, and errors ironed out. During each iteration, developers would (re)compile, undeploy the previous installation, and then (re)deploy the new Web application.

In addition to these tasks, all other actions that can be executed from the Web UI to the manager, such as viewing resources, roles, server information, and so on, can be performed via the Ant script, too:

```
<target name="resources" description="Tomcat resources">
        <!-- A 'type' attribute can query specific resources -->
        <resources url="${tomcat.manager.url}" username="${tomcat.username}"
                password="${tomcat.password}"/>
</target>

<target name="roles" description="Tomcat roles">
        <roles url="${tomcat.manager.url}" username="${tomcat.username}"
                password="${tomcat.password}"/>
</target>

<target name="serverinfo" description="Server info">
        <serverinfo url="${tomcat.manager.url}" username="${tomcat.username}"
                password="${tomcat.password}"/>
</target>

<target name="sessions" description="Web application sessions">
        <sessions url="${tomcat.manager.url}" username="${tomcat.username}"
                password="${tomcat.password}" path="${context.path}"/>
</target>
```

In addition, there are JMX tasks that allow for querying, viewing, and modifying Tomcat's behavior at runtime. In the example of this that follows, the `jmxget` target shows the value of the maximum active sessions attribute for the `hello` Web application, and the `jmxget` target sets this property to `100`. The `jmxquery` target shows all servlets loaded by the Tomcat container.

```
<!-- Sample JMX get call: Show the maximum active sessions settings for
         the 'hello' web application -->
<target name="jmxget" description="JMX Get command">
        <jmxget url="${tomcat.manager.url}"
                username="${tomcat.username}" password="${tomcat.password}"
                bean="Catalina:type=Manager,path=/hello,host=localhost"
```

(continued)

```
                attribute="maxActiveSessions" />
</target>

<!-- Sample JMX query: show all servlets loaded by the Tomcat container -->
<target name="jmxquery" description="JMX Query command">
        <jmxquery url="${tomcat.manager.url}"
                username="${tomcat.username}" password="${tomcat.password}"
                query="*:j2eeType=Servlet,*" />
</target>

<!-- Sample JMX get call: Set the maximum active sessions settings for
          the 'hello' web application to 100 -->
<target name="jmxset" description="JMX Set command">
        <jmxset url="${tomcat.manager.url}"
                username="${tomcat.username}" password="${tomcat.password}"
                bean="Catalina:type=Manager,path=/hello,host=localhost"
                attribute="maxActiveSessions" value="100"/>
</target>
```

JMX is covered in more detail in Chapter 16.

Known Issue: Failure While Undeploying Web Applications on Windows

Undeploying a Web application on Windows may sometimes fail because JAR files under `WEB-INF/lib` haven't been deleted. The cause of this problem is an "optimization" that causes the JVM to aggressively hold on to the JAR and resource files. This can be an issue in development environments where *hot deployments* (i.e., deploy/undeploy without restarting Tomcat) are useful.

This is a workaround for this issue: Add the following as the Web application content under `META-INF/context.xml`:

```
<context antiJARLocking="true" antiResourceLocking="true">
   ...
</context>
```

This workaround is not without its side effects, however:

- Setting `antiJARLocking` to `true` forces the JVM to take extra measures to prevent JAR file locking, such as copying the files to a temporary directory. This will negatively affect the startup time of Web applications.
- Setting `antiResourceLocking` to `true` will similarly result in much slower startup times of Web applications. At the time of this writing, setting this to `true` also triggers a Tomcat bug that prevents JSP page reloading.

The change to the context can be made across all Web applications by adding it to `<TOMCAT_HOME>/conf/context.xml`.

Tomcat Manager — Using HTTP Requests

Finally, another way to use the Tomcat manager is by issuing an HTTP request to specific URLs. The manager application commands that are issued via the Web browser have the following format:

```
http://{hostname}:{portnumber}/manager/{command}?{parameters}
```

In this command, the various parts are as follows:

- `hostname`: The host on which the Tomcat instance is running.
- `portnumber`: The port on which the Tomcat instance is running.
- `command`: The manager command to be run. The allowed values for `command` are `list`, `sessions`, `start`, `stop`, `install`, `remove`, `deploy`, `undeploy`, `reload`, `serverinfo`, `roles`, `resources`, `status`, `jmxget`, `jmxset`, and `jmxproxy`.
- `parameters`: The parameters passed to the commands listed previously. These are command-specific, and are explained in detail along with the specific command in the following list. Many of these parameters contain the context path to the Web application (the `path` parameter) and the URL to the Web application file (the `war` parameter). The context path for the `ROOT` application is an empty string. For all other Web applications, the context path must be preceded by a slash (`/`). The URL to the Web application can be in one of the following formats:
  - `file:/absolute/path/to/a/directory`: This specifies the absolute path to a directory in which a Web application is present in an unpackaged form. This entire path is then added as the context path of the Web application in Tomcat's configuration.
  - `file:/absolute/path/to/a/webapp.war`: This specifies the absolute path to a WAR file. The Tomcat documentation states that this format is not allowed for the `install` command. However, our tests with Tomcat 4.1.3 indicate that it works fine for `install`, too.
  - `jar:file:/absolute/path/to/a/warfile.war!/`: The `jar` protocol enables the specifying of the URL for a WAR file. This is handled by the `java.net.JarURLConnection`, which provides a URL connection to a JAR/WAR file. Here, the URL specified is for a file on the local file system.
  - `file:/absolute/path/to/a/context.xml`: This specifies the absolute path to the context configuration file. This is an XML file that contains the `<Context>` configuration for the Web application.
  - `directory`: This is the directory name for the Web application within the Tomcat application base directory. The application base directory is typically `$CATALINA_HOME/webapps`.
  - `webapp.war`: This is the directory name for the Web application archive (that is, WAR file). This WAR file is looked for in the Tomcat application base directory (typically, `$CATALINA_HOME/webapps`).

The `!/` characters at the end of these URLs enable them to be used in a Web browser and not cause the default MIME type action for the `.war` extension to take effect. For example, if the following URL is used to install a Web application and the `!/` is omitted at the end, the user may be prompted to (depending on

how the browser's MIME settings are configured) save to disk, open the file in the browser, or open the file in an application (for example, Winzip):

```
http://localhost:8080/manager/install?path=/hello&war=jar:file:/path/to/hello.war!/
```

Numerous problems can occur while working with the manager application. The possible causes of failure are listed later in this chapter in the section "Possible Errors."

List Deployed Applications

The format for the `list` command URL that lists all deployed and installed applications is as follows:

```
http://{hostname}:{portnumber}/manager/list
```

Figure 8-5 shows the `list` command being run.

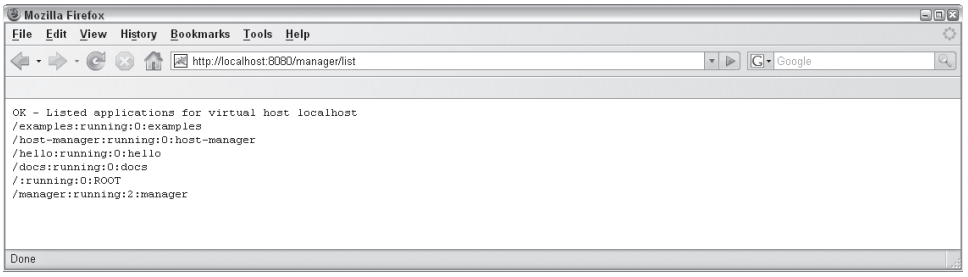

Figure 8-5: Listing deployed applications

As shown in the figure, the response for a successful command execution begins with an `OK` string (`OK -- Listed applications of virtual host {hostname}`). A missing `OK` is an indication of failure, and the rest of the response page provides the cause(s). The possible causes of failure for each command are covered later in the chapter. The response page is in the `text/plain` format (that is, it contains no HTML markup).

The data fields returned in a manager command response are always delimited by the colon (:) character. In Figure 8-5, each line indicates the (unique) context path of the Web application, the status (running or stopped), the number of active sessions for the application, and the document base for the Web application.

These conventions enable scripts to be written that retrieve the output of the `manager` command and perform appropriate actions.

Deploying a New Application

The `deploy` command is used to deploy a Web application to a running instance of Tomcat. The effect of this command is as follows:

- The WAR file for the Web application is uploaded from the client machine to the machine on which Tomcat is running, and copied into the application base directory of the given virtual host.

For example, if the virtual host name configured in `server.xml` were `localhost` itself, the WAR file would be copied under `$CATALINA_HOME/work/Standalone/localhost/manager`.

- The client machine may very well be the same machine on which Tomcat is running.
- An entry for the Web application's context is added into Tomcat's runtime data structures.
- The Web application is loaded.
- The WAR file can contain a `<Context>` element definition (`META-INF/context.xml`), and this would take precedence over the default `Context` that the `manager` application generates for it.

The general format for the `deploy` command is as follows:

```
http://{hostname}:{portnumber}/manager/deploy?path={context_path}
```

Here, *`hostname`* and *`portnumber`* are the host and port for the Tomcat instance, and *`context_path`* is the context path for the application. The WAR file to be deployed is passed inside the request data of the HTTP `PUT` request. Therefore, to deploy the hello application shown at the beginning of the chapter at the context path `/hello`, an HTTP `PUT` request would need to be directed to the URL `http://{hostname}:{portnumber}/manager/deploy?path=/hello`.

Because the WAR file is passed as the request data, this command cannot be invoked directly via a Web browser. Instead, it should be invoked from a tool — for example, within an Ant script (see the section "Tomcat Manager: Managing Applications with Ant," earlier in this chapter) or via the manager Web interface (see the section "Tomcat Manager: Web Interface," earlier in this chapter).

Essentially, the tool that sends the `deploy` command will have to build an HTTP request that looks something like the one shown in Figure 8-6, and execute an HTTP `PUT` command to send it over to the `manager` servlet.

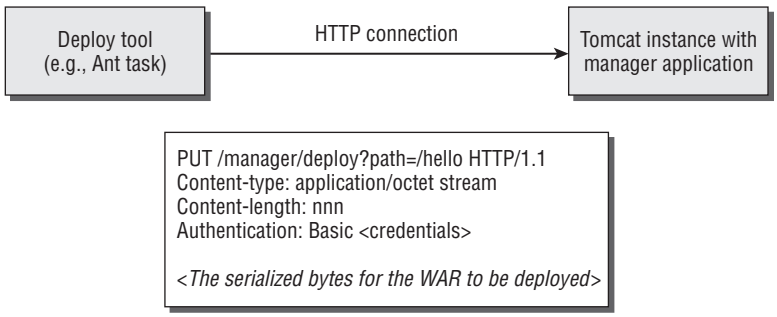

Figure 8-6: Deploy command sent over an HTTP connection

A successful `deploy` command returns the success message, `OK -- Deployed application at context path {context_path}`. If the operation failed, the error message would start with a `FAIL` string and contain the cause for failure.

Installing/Deploying Applications in Tomcat 6

Tomcat 4.*x* had both an `install` command and a `deploy` command. At first glance, these look the same. However, there were differences as far as the Tomcat `manager` application is concerned.

When a Web application is deployed, it makes permanent changes to Tomcat's configuration, so the Web application is available across Tomcat restarts. The `install` option, however, did not make permanent changes to Tomcat's configuration. Thus, the `install` command was useful for test purposes. Developers can build a Web application, install it, and then try it out. Once it is sufficiently robust, it can then be deployed via the `deploy` command to permanently place it into a Tomcat installation.

Beginning with Tomcat 5, the confusion around the `install/deploy` commands was simplified by having just one command: `deploy`. Correspondingly, the `uninstall` command was deprecated in favor of the `undeploy` command.

The `deploy` command enables Web applications to be deployed either from the local file system (local to the machine on which Tomcat is running) or remotely. The deployment changes are permanent (that is, they survive Tomcat restarts) until the application is undeployed.

Deploying a New Application Remotely

The minimum format for the remote `deploy` command is as follows:

```
http://{hostname}:{portnumber}/manager/deploy?path={context_path}
```

In addition to this, the (remote) `deploy` command can take three additional and optional parameters:

- `update`: When set to `true`, the previously deployed instance of the Web application will first be undeployed, and then the new one deployed. This defaults to `false`.
- `tag`: Assigns a version tag to the deployed Web application. This enables versioning of Web applications, and a subsequent redeployment of the Web application using only this version tag.
- `pause`: Causes the Web application to be paused during the deployment so that incoming requests are not lost. The `pause` property defaults to `false`, and is used in conjunction with the `update` property.

Thus, deploying the `hello` Web application with default parameters and a tag of `hello_ver1` could look like the following:

```
http://localhost:8080/manager/deploy?path=/hello&update=false&tag=hello_
ver1&pause=false
```

Because the WAR file is passed as the request data (via an HTTP `PUT`), this command cannot be invoked directly via a Web browser. Instead, it should be invoked from a tool — for example, within an Ant script (see the section "Tomcat Manager: Managing Applications with Ant," later in this chapter) or via the manager Web interface (see the section "Tomcat Manager: Web Interface," earlier in this chapter).

Deploying a New Application from a Local Path

The `deploy` command also enables the installation of a Web application from a local path. That is, the Web application is present either as a WAR file or a directory on the same machine on which Tomcat is installed. The directory from which the Web application is installed must, as always, have a directory structure corresponding to the conventions of a Web application.

A lot of combinations are possible while deploying from a local path. These enable you to do the following:

- Install from a local directory or WAR file (anywhere on the file system)
- Install from a local directory or WAR file within the Tomcat application base
- Install using a Context configuration file
- Redeploy a previously deployed version of a Web application

If installation is being done from a file system location outside the Tomcat application base (`$CATALINA_HOME/webapps`), the fully qualified path to the WAR file or directory needs to be specified.

For example, the following is used to install from a directory:

```
http://localhost:8080/manager/deploy?path=/hello&war=file:/path/to/hello
```

Alternatively, the following is used from a WAR file:

```
http://localhost:8080/manager/deploy?war=jar:file:/path/to/hello.war!/
```

Figure 8-7 shows a deployment of a WAR file (`C:\hello.war`) with the tag `hello_ver1`.

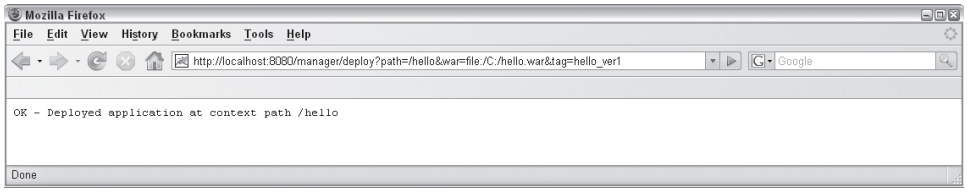

Figure 8-7: Deploying a Web application from a local path in Tomcat 6

In the current version of Tomcat 6, a `local path` *deployment, as shown in Figure 8-7, does not copy the WAR files into the Tomcat application base and, hence, the deployment does not persist across Tomcat restarts. Deployment on the local machine can also be done by copying the WAR file (or directory) to the application base. In this case, the deployment would naturally persist across Tomcat restarts.*

Another way to deploy is to first copy the directory or WAR file over to the Tomcat application base directory. If the `autoDeploy` attribute in the Host (see `server.xml`) had been set to `true` (that is, the default), the Web application would be automatically deployed once it was copied over. Because this is turned off for performance reasons (at least it should be), the administrator would need to deploy the Web application explicitly. As the directory or WAR file is copied to the application base directory (`$CATALINA_HOME/webapps`), the full pathname for the WAR file does not need to be specified (see Figure 8-8):

```
http://localhost:8080/manager/deploy?path=/hello&war=hello.war
```

In this command, the Web application would be accessible under the context named `/hello`. If it is to be deployed under another context name (such as `/aloha`), it could be specified as follows:

```
http://localhost:8080/manager/deploy?path=/aloha&war=hello.war
```

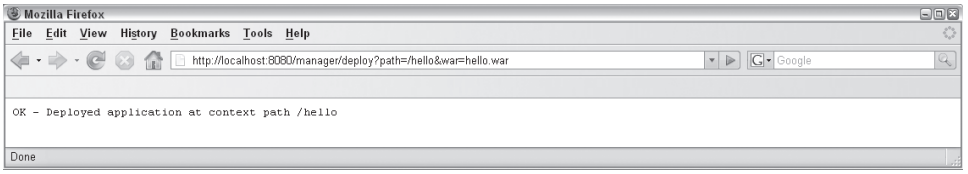

Figure 8-8: Deploying a Web appliction from a WAR file in Tomcat 6

In each of these cases, the WAR file or the expanded directory can contain a `context.xml` file with a `Context` entry for the Web application. The context file can, however, be overridden by specifying it in the following command:

```
http://localhost:8080/manager/deploy?config=file:/path/context.xml&war=jar:file:/
path/hello.war!/
```

Finally, installation can be done via a `context.xml` file:

```
http://localhost:8080/manager/deploy?config=file:/path/context.xml
```

The `context.xml` in this case would contain the details for the Web application — the context name (`/hello`) and the `docBase`:

```
<Context path="/hello" docBase="/path/to/hello"
         debug="0">

  <!-- Link to the user database we will get roles from -->
  <ResourceLink name="users" global="UserDatabase"
                type="org.apache.catalina.UserDatabase"/>

</Context>
```

Using a `context.xml` *file is subject to the* `deployXML` *flag in the Host element (*`server.xml`*). If this is set to* `false` *(the default is* `true`*), applications cannot be installed via a context definition. Nor can they be installed outside the Host's config base directory (*`$CATALINA_HOME/conf/[engine_name]/[host_name]`*).*

If a Web application has already been deployed with a version tag (either remotely or locally), it can be redeployed using the tag name. This is very useful during development because it enables rolling back to an older version:

```
http://localhost:8080/manager/deploy?path=/hello&tag=hello_ver1
```

Another Host attribute to note for deploying Web applications is `unpackWARs`*. This controls whether the WAR is unpacked into a directory (*`unpackWARs=true`*), or the Web application is run from a WAR file itself (*`unpackWARs=false`*).*

Reloading an Existing Application

An existing application can be reloaded by accessing the `manager` application via the following URL:

```
http://{hostname}:{portnumber}/manager/reload?path={context_path}
```

This causes the existing application to shut down and then restart. The application's deployment descriptor (`web.xml`) is not reread (at least not in the current version of Tomcat), even though the Tomcat documentation states that it is. This is a known bug, and it is expected that a future version of Tomcat will fix it. The workaround is to stop and then start the application again. The `server.xml` configuration file is not reread either, but this is by design.

The `reload` command is useful when a Web application has not been configured to be reloadable. A Web application's `<Context>` entry in the `server.xml` file has a `reloadable` attribute. When this attribute is set to `true`, Tomcat monitors all its classes in `/WEB-INF/classes` and `/WEB-INF/lib` and reloads the Web application if a change is detected. This causes a performance hit in production environments, as the class loader keeps comparing the date and time stamps for servlets in memory with those on disk. To avoid this, the `reload` command can be used to make Tomcat reload the Web application when developers change any classes.

The standard Java class loader is designed to load a Java class just once. So how does the `reloadable` attribute work? Tomcat implements its own custom class loader that is used to reload the classes in `/WEB-INF/classes` and `/WEB-INF/lib` if required. Chapter 9 discusses this topic in more detail.

> *The current version of Tomcat supports reloading only if a Web application has been installed from an unpacked directory. It does not support reloading if the Web application has been installed from a WAR file. The workaround with a WAR file is to either restart Tomcat or remove and then deploy the application again.*

Figure 8-9 shows the `hello` Web application being reloaded.

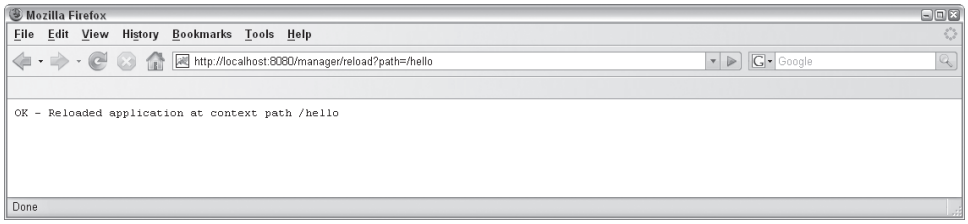

Figure 8-9: Reloading a Web application

A successful execution of the `reload` command returns an `OK -- Reloaded application at context path {`*`context_path`*`}` message, where {*`context_path`*} is the context path for the application.

Listing Available JNDI Resources

The general format of the URL for listing available JNDI resources is as follows:

```
http://{hostname}:{portnumber}/manager/resources[?type={jndi_type}]
```

In this URL, the `type` argument is optional. When it is not specified, all the available JNDI resources are listed. Otherwise, JNDI resources corresponding to the specified type alone are listed. The `type` field

needs to be a fully qualified Java class name. For example, for JDBC data sources, the type needs to be specified as `javax.sql.DataSource`:

```
http://localhost:8080/manager/resources?type=javax.sql.DataSource
```

The response to this contains a success string (`OK -- Listed global resources of all types` or `OK -- Listed global resources of type {jndi_type}`), followed by information about the resources (one per line). Each line contains the global resource name and the global resource type. The global resource name is the name of the JNDI resource as specified in the `global` attribute of the `<ResourceLink>` element in Tomcat's configuration. The global resource type is the fully qualified Java class name of this JNDI resource (see Figure 8-10).

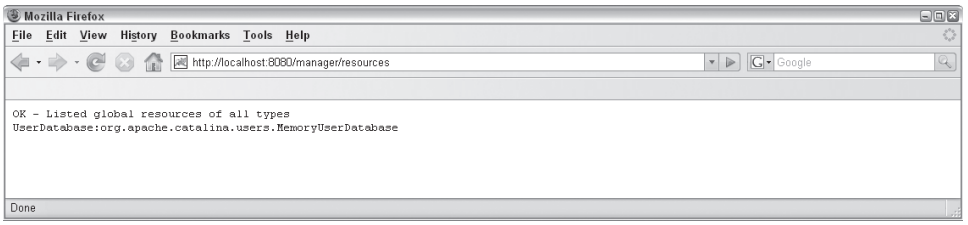

Figure 8-10: Listing JNDI resources

Listing OS and JVM Properties

The URL for listing these properties is as follows:

```
http://{hostname}:{portnumber}/manager/serverinfo
```

Figure 8-11 shows the `serverinfo` command being run. It displays the OS name, the OS version, and information about the Java Virtual Machine (JVM) being used.

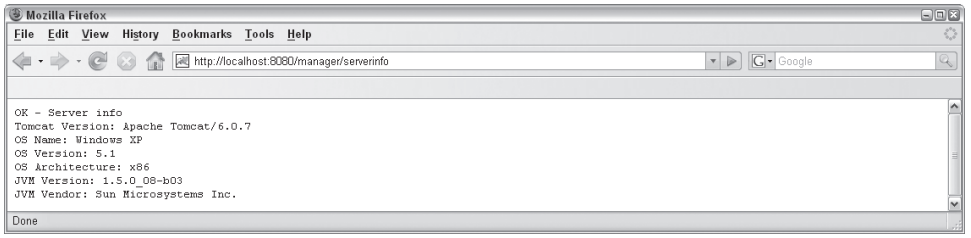

Figure 8-11: Listing OS and JVM properties

Stopping an Existing Application

The `manager` application can be used to stop a running application. The following URL shows how this can be done:

```
http://{hostname}:{portnumber}/manager/stop?path={context_path}
```

This command sends a signal to the Web application to stop. This application is no longer available to users, although it still remains deployed. If the `list` command is run again, the state of the application is shown as "stopped" (see Figure 8-12).

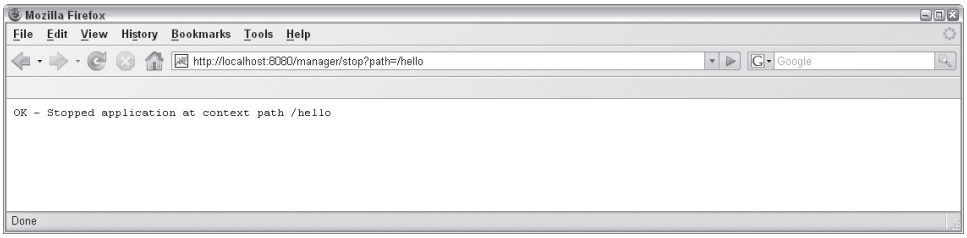

Figure 8-12: Stopping a Web application

If the application stops successfully, the message `OK -- Stopped application at context path {context_path}` is displayed. If the operation fails, a `FAIL` message with appropriate error information is shown. Stopping a Web application does not affect any Tomcat configuration information kept on the file system, so if Tomcat is restarted, the application is started, too.

The application can be restarted using the `start` command.

Starting a Stopped Application

The `manager` application can be used to start a stopped application. The following URL shows how this can be done:

```
http://{hostname}:{portnumber}/manager/start?path={context_path}
```

Here, {*context_path*} is the context path for the Web application (an empty string for the `ROOT` application).

If the application starts successfully, the message `OK -- Started application at context path {context_path}` is displayed (see Figure 8-13).

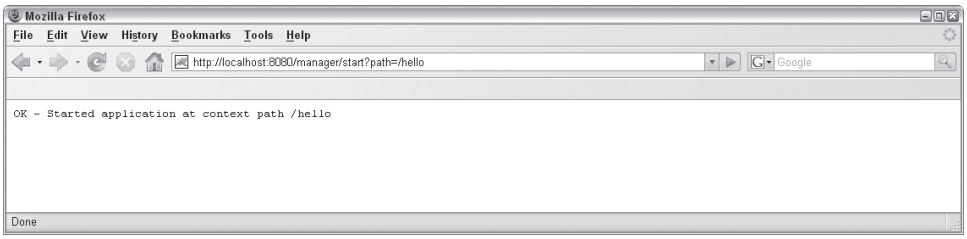

Figure 8-13: Starting a stopped Web application

If the operation fails, a `FAIL` message with appropriate error information is displayed.

Undeploying a Web Application

This command first signals the application to shut down (if it is still running) and then deletes the Web application directory and the application WAR file. It then removes the `<Context>` entry for the Web application from `$CATALINA_HOME/conf/server.xml`.

In short, the `undeploy` command does the opposite of the `deploy` command described earlier in the chapter. However, the `undeploy` command works only on applications installed in the application base directory of the virtual host (the location where the `deploy` command put the Web application WAR files or the extracted directories).

> *This command should be used with care. It deletes the Web application directory that was created when the application was deployed. In Tomcat 4.x, there is an option to use the* `remove` *command instead, which does not remove the Web application permanently but only for the current Tomcat lifetime.*

The URL for the `undeploy` command is as follows:

```
http://localhost:8080/manager/undeploy?path={context_path}
```

If the application undeploys successfully, the message `OK -- Undeployed application at context path {context_path}` is displayed, as shown in Figure 8-14. If the operation fails, a `FAIL` message with the appropriate error information is displayed.

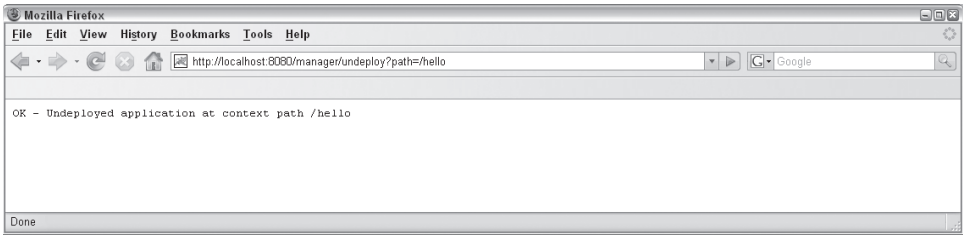

Figure 8-14: Undeploying a Web application

Older versions of Tomcat (Tomcat 4.0 and before) have a `remove` command that has since been deprecated in favor of `undeploy`.

Displaying Session Statistics

The `manager` application can be used to retrieve statistics about a particular Web application. The statistics shown are the default session timeout and the number of current active sessions.

The URL for accessing this information is as follows:

```
http://localhost:8080/manager/sessions?path={context_path}
```

For example, the statistics for the `hello` application can be checked using the following command (see Figure 8-15):

```
http://localhost:8080/manager/sessions?path=/hello
```

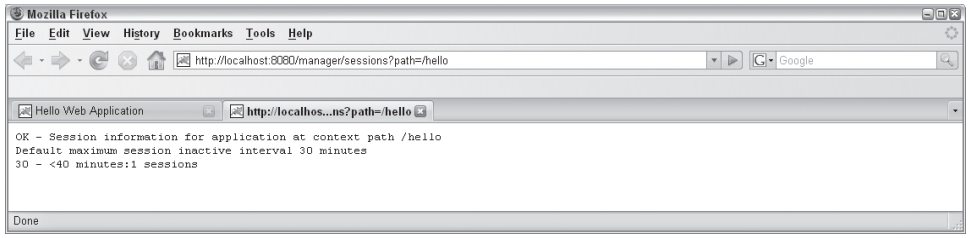
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/manager/sessions?path=/hello
Hello Web Application
http://localhos...ns?path=/hello
OK - Session information for application at context path /hello
Default maximum session inactive interval 30 minutes
30 - <40 minutes:1 sessions
Done

Figure 8-15: Displaying session statistics

Querying Tomcat Internals Using the JMX Proxy Servlet

The JMX `proxy` servlet enables the querying of Tomcat internals classes (or any other class exposed via MBeans). The general format of this command is as follows:

```
http://{hostname}:{portnumber}/manager/jmxproxy/?qry=QUERY_STRING
```

A missing query string (that is, only `http://{hostname}:{portnumber}/manager/jmxproxy/`) will show all the MBeans (a long listing!). The query parameters are not well documented and some experimentation is required to see what works. For example, a query string `qry=*:j2eeType=Servlet,*` shows all loaded servlets. As a browser URL, this would be written as follows (see Figure 8-16):

```
http://localhost:8080/manager/jmxproxy/?qry=*%3Aj2eeType=Servlet%2c*
```

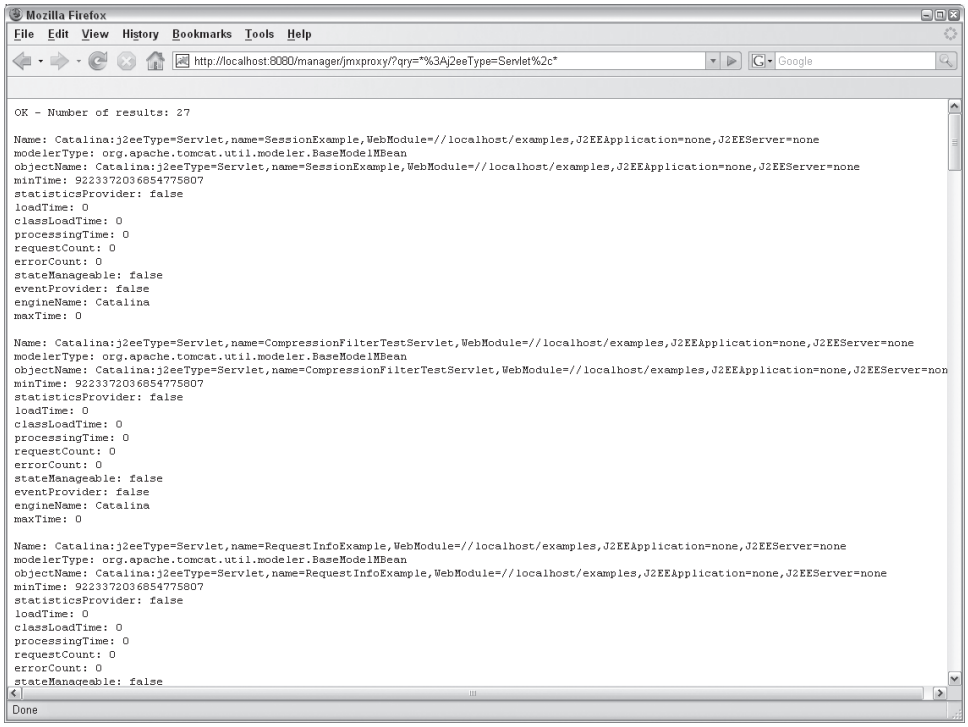

Figure 8-16: Querying Tomcat internals using the JMX proxy servlet

The % values in the command are hexadecimal escape sequences for reserved characters used in URLs (as defined in RFC 2396): %3A for the colon (:), %2C for the comma (,), and %3D for the equal sign (=). When this command is run via the Ant task interface (see the section "Tomcat Manager: Managing Applications with Ant," earlier in this chapter), the escape sequences don't have to be used.

The JMX proxy servlet also allows for changes to these values.

Setting Tomcat Internals Using the JMX Proxy Servlet

The general format for the JMX set command is as follows:

```
http://{hostname}:{portnumber}/manager/jmxproxy/set=BEANNAME&att=
MYATTRIBUTE&val=NEWVALUE
```

Here, BEANNAME is the bean name, MYATTRIBUTE is the name of the bean attribute that needs to be modified, and NEWVALUE is the new value for the bean attribute.

As shown in Figure 8-17, the following command sets the maximum number of active sessions to 100 in a running Tomcat Web application (set the maxActiveSessions attribute in the bean Catalina:type=Manager,path=/hello,host=localhost to 100):

```
http://localhost:8080/manager/jmxproxy/?set=Catalina%3Atype%3DManager%2Cpath%3D/
hello%2Chost%3Dlocalhost&att=maxActiveSessions&val=100
```

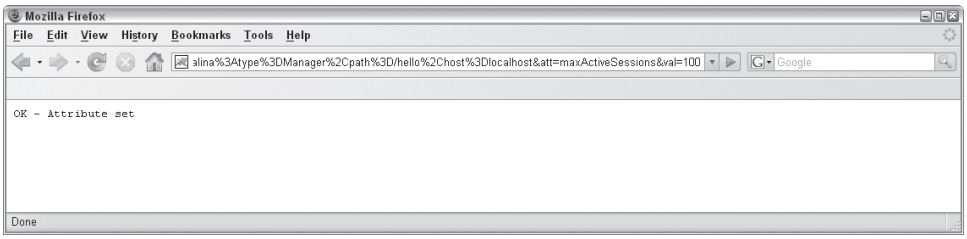

Figure 8-17: Setting Tomcat internals using the JMX proxy servlet

As before, the % values in the command are hexadecimal escape sequences for reserved characters. When this command is run via the Ant task interface (see the section "Tomcat Manager: Managing Applications with Ant," earlier in this chapter), the escape sequences don't have to be used.

Chapter 16 discusses the JMX support in Tomcat in great detail.

Possible Errors

Numerous things can go wrong while working with the manager application. The following list describes some of the typical error messages and the possible causes of failure. These errors are applicable for the HTTP command interface, the Web application interface, and the Ant task interface to the manager application commands.

- **Application already exists at path** `{context_path}`**:** A Web application already exists at the path specified. The context path for each Web application must be unique. Tomcat returns an error if there is another application with the same context path. This can be the same application (that is, `deploy` was executed twice for the same application) or a different one with the same context path. To fix this, the previous application must be `undeployed/removed`, or a different context path chosen.
- En**countered exception:** An exception occurred while trying to start the Web application. The Tomcat log files will contain error messages relating to the specific error. Typical causes of this error are missing classes/JAR files while loading the application, and invalid commands in the application's `web.xml` file.
- **Invalid context path specified:** The context path must start with a slash (`/`). The exception to this is when the `ROOT` Web application (that is, at context path `/` itself) is being deployed, in which case the context path must be a zero-length string.
- **No context path specified:** The context path is mandatory.
- **Document base does not exist or is not a readable directory:** The value specified for the WAR file path/URL in the `war` parameter is incorrect. This parameter must point to an expanded Web application or an actual WAR file.
- **No context exists for path** `{context_path}`**:** The context path is invalid; there is no Web application deployed that corresponds to it.
- **Reload not supported on WAR deployed at path** {context_path}**:** The Web application was installed from a WAR file, instead of from an unpacked directory. The current version of Tomcat does not support this.
- **No global JNDI resources:** No JNDI global resources were configured for this Tomcat instance.
- **Cannot resolve user database reference:** There was an error looking up the appropriate user database. For example, in the case of the roles stored in a JNDI Realm, a JNDI error would result in such a message. Tomcat's log files would have more error information.
- **No user database is available:** The `<ResourceLink>` element has not been configured properly in the `manager.xml` configuration file. See the section "Manager Application Configuration" earlier in this chapter for more information.

The error messages shown here are in English, but Tomcat supports numerous languages. The locale-specific versions of these messages (error as well as the success messages) are picked up from resource bundles.

Security Considerations

Securing the `manager` application is critical. An insecurely configured manager could be used to cause denial of service by stopping existing applications, or worse, to install malicious Web applications over existing ones. The following are some of the ways in which the `manager` application can be secured:

- Use a more rigorous mechanism of authentication for the `manager` application than `BASIC`. Administrators can, for example, set the authentication method to be client-certificate–based (`CLIENT-CERT`) in the deployment descriptor. `BASIC` authentication is very insecure because the

password is sent across as a standard base64-encoded string. `CLIENT-CERT` uses SSL for securing the transport layer. The manager deployment descriptor configured for `CLIENT-CERT` is as follows:

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>Tomcat Manager Application</realm-name>
</login-config>
```

- Use JDBC or JNDI-based Realm implementations to store the manager user name/password. These are more secure than Memory/UserDatabase Realms because they don't save the password in a text file on the file system. This can be a security risk if the file permissions aren't set correctly, although similar problems can occur with JDBC/JNDI Realms, too, if the access rights to the database, LDAP server, and so on, were too permissive. These Realms are covered in more detail in Chapter 14. If you are using another Realm, remember to update the `manager.xml` file with the Realm the roles are to be picked up from.
- Configure the Realm implementation to use encrypted passwords. This is especially useful if Memory or UserDatabase Realm implementations are used. Two sample configurations of this are as follows:
  - The `server.xml` file:

```
<Realm className="org.apache.Catalina.realm.UserDatabaseRealm"
           debug="5"
           digest="sha"
           pathname="conf/tomcat_users.xml" />
```

  - The `tomcat-users.xml` file:

```
<user name="manager"
      password="c23e4c2003a93af2dad4dae78f5e1c4a4735732"
      roles="manager" />
```

- The character sequence in the `password` attribute is the SHA digest version of the password.
- Use a `RemoteAddrValve` or `RemoteHostValve` Valve in the manager's Context to restrict the machines from which the manager application can be accessed. In the following example, access is restricted to the host on which Tomcat runs by using the loopback IP address (127.0.0.1):

```
<Context antiResourceLocking="false" privileged="true">

          <Valve className="org.apache.Catalina.valves.RemoteAddrValve"
                 allow="127.0.0.1"/>
</Context>
```

- The `deployXML` parameter in the configuration for a Host (see `server.xml`) controls whether Web applications can be deployed using a context configuration file, and whether they can be installed outside the Host's config base directory (`$CATALINA_HOME/conf/[engine_name]/[host_name]`). This is set to `true` by default (that is, allowing an install outside the config base). Setting it to `false` can prevent users from deploying Web applications anywhere else

on the file system, where the admin may not be able to control file permissions, as you can see in the following example:

```
<Host name="localhost"
 deployXML="false"
 debug="0"
 appBase="webapps"
 unpackWARs="true"
 autoDeploy="false">
        ...
 </Host>
```

Tomcat Deployer

Tomcat comes with a "deployer" distribution that can be downloaded from the Apache Web site. These distributions are named as `apache-tomcat-6.x.y-deployer.tar.gz` or `apache-tomcat-6.x.y-deployer.zip`, where `x.y` is the Tomcat version.

The deployer distribution is a stripped-down version of Tomcat that includes the following:

- Tomcat's Ant tasks for managing the Web application. These tasks are discussed in detail earlier in the chapter.
- A sample build file (`build.xml`) for deploying a Web application. This build file uses a property file (`deployer.property`) that can be modified as per requirements.
- A JSP compiler (Jasper) for precompiling JSP pages.

The deployer distribution can be used to validate, compile, deploy, and manage Web applications, and includes only those parts of Tomcat that are required for this.

Summary

The `manager` application provides an easy-to-use interface (via the Web-based interface), and enables the automation of tasks (for example, via the `manager` application's Ant tasks). This chapter covered issues related to the `manager` Web application, including the following:

- Configuration for the `manager` application
- Administration capabilities of the Tomcat `manager` application
- Security-related issues

Securing the `manager` application is important. As discussed earlier, someone who gains unauthorized access to applications can do a lot of harm, such as deploy malicious applications or cause a Denial of Service (DoS) by shutting down running ones. Administrators concerned about security should perform the appropriate configuration as specified in the section "Security Considerations," or, if they are paranoid, disable the `manager` Web application altogether from production deployments.

Chapter 9 discusses advanced Java class loaders and their relevance to Tomcat administration.

9

Class Loaders

Every Java developer makes extensive use of class loaders, often without realizing it. Each time a class is instantiated as an object or referenced statically, that class must be loaded by the Java Virtual Machine (JVM) into memory. Thus, even statements as simple as `String greeting = "hello"` or `int maxValue = Integer.MAX_VALUE` make use of a class loader. They require the `String` class and the `Integer` class to be loaded, respectively.

While class loaders are designed to operate fairly transparently from the developer's point of view, there are subtleties to their use that are important to understand. Why a chapter on class loading in a Tomcat book? It turns out that class loaders and their behavior are a big part of Tomcat. Following the Servlet specification, Tomcat is required to allocate a unique class loader to each Web application. This chapter explains what this means and why it is important.

Following an explanation of class loaders in general and Tomcat's class loaders in particular, we discuss common problems related to class loaders. By the end of this chapter, not only will you be familiar with class loaders in general, but you'll also understand how they relate specifically to Tomcat.

The following topics are covered in this chapter:

- An overview of class loaders
- Security issues with class loaders
- Tomcat and class loaders
- Dynamic class reloading
- Common class-loader issues

Class Loader Overview

Java was designed to be platform-independent and to support distributed network architectures. To fulfill both of these goals, Java had to innovate in many key areas. One of these areas is the basic issue of how to load code libraries. If Java is to be truly platform-independent, it cannot rely on a

specific type of file system (or even a set of dozens of file systems) for loading its libraries. Many small embedded computer systems don't even have a file system!

Furthermore, because Java was designed to load classes from various sources spread across a network, simply loading classes from a file system won't work.

To deal with these issues, the Java architects introduced the notion of a *class loader*. The role of the class loader is to abstract the process of loading classes, making it completely independent of any type of underlying data store, be it a network or a hard drive.

For example, consider the following simple program:

```
import com.wrox.MyObject;
public class Simple {
    public static void main(String[] args) {
        MyObject myObject = new MyObject();
    }
}
```

When the line `MyObject myObject = new MyObject()` is executed, the Java Virtual Machine (JVM) asks a class loader to find a class named `com.wrox.MyObject` and return it as a `Class` object. The class loader is then free to do whatever it is designed to do to locate the class. Possible actions include searching a file system, checking a ROM chip, or loading a class from a network. Once returned, the `Class` object that represents the `MyObject` class is then used to instantiate the `myObject` instance. Figure 9-1 depicts this process.

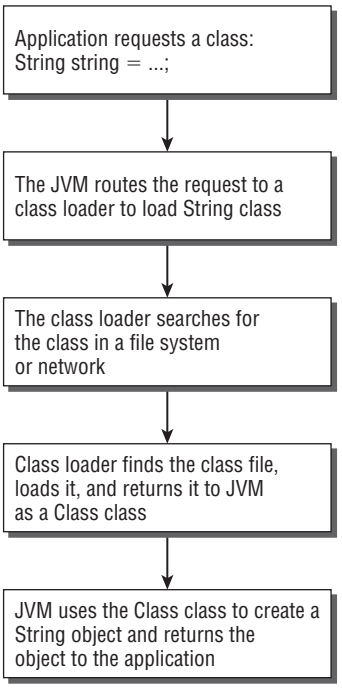
Application requests a class:
String string = ...;
The JVM routes the request to a class loader to load String class
The class loader searches for the class in a file system or network
Class loader finds the class file, loads it, and returns it to JVM as a Class class
JVM uses the Class class to create a String object and returns the object to the application

Figure 9-1: JVM and class loaders

Standard Java SE Class Loaders

Ever since the J2SE 1.2 specification, the JVM has made use of three distinct class loaders, which are discussed next, along with their roles:

- Bootstrap class loader (also called the *primordial* class loader)
- Extension class loader
- System class loader

These class loaders occupy a hierarchy, with the system class loader at the bottom and the bootstrap class loader at the top. The relationships are parent-child, so the parent of the system class loader is the extension class loader. The importance of this relationship will soon become clear.

Bootstrap Class Loader

As its name implies, the bootstrap class loader is used by the JVM to load those Java classes that are necessary for the JVM to function. Actually, the bootstrap class loader is responsible for loading *all* the core Java classes (such as `java.lang.*` and `java.io.*`).

Because class loaders are written in Java, the bootstrap class loader solves a high-tech "chicken-and-egg" problem: How can the JVM load a Java-based class loader when the class loader itself must be loaded? Including the bootstrap class loader in the JVM itself solves this problem, and various JVM vendors (including Sun) implement the bootstrap class loader using native code.

Although it has been explained that the bootstrap class loader loads the core Java classes, you may be wondering where exactly the bootstrap class loader finds these classes. It turns out that exact JAR files vary from vendor to vendor. Sun's Java SE 5 JVM for instance looks in JAR files under `<JDK>/jre/lib`, particularly in `<JDK>/jre/lib/rt.jar` for bootstrap classes. IBM's JVM on the other hand, reads these bootstrap classes from `<JDK>/jre/lib/vm.jar`. Here, `<JDK>` refers to the install location of the Java 5 JDK.

Both Sun's and IBM's JVM also have non-standard command line options: `-Xbootclasspath`, which allows you to set a different location for the JVM to look for the bootstrap classes, and `-Xbootclasspath/p:` and `-Xbootclasspath/a:`, which allow you to prepend or append a `CLASSPATH` to the boot `CLASSPATH`. For Web application development, there should never be a reason for you to change or add to the boot `CLASSPATH`.

Extension Class Loader

Java 1.2 introduced the standard extension mechanism. Normally, when developers want the JVM to look in certain locations for class files, they make use of the `CLASSPATH` environment variable. Sun introduced the standard extension mechanism as an alternative method. You can drop JAR files into a standard extension directory and the JVM will automatically find them.

The extension class loader is responsible for loading all the classes in one or more extension directories. Just as the bootstrap class loader's paths can vary on different JVMs, so can the standard extension paths. The standard extension directory for Sun's and IBM's JDK on Linux is `<JDK>/jre/lib/ext`. Here `<JDK>` refers to the install location of the Java 5 JDK. If only the Java Runtime Environment (JRE) is installed, then the extension directory is `<JRE>/lib/ext`, where `<JRE>` is the install location of the Java runtime. The Extension class loader need not be a separate class loader — some implementations might even have the bootstrap class loader load up the classes from the extension directory.

The standard extension mechanism is usually used by the JVM for optional JAR files that are not a part of the core JARs, but yet need to be available to all Java applications. The Java Cryptography Extension (JCE) JAR files are an example of a type of JARs that are typically put in the extension directory. Any JAR files that are added to the extension directory don't need to be listed on the `CLASSPATH`. However, you should be careful about adding JAR files to the extension directory as these JARs are visible to all applications using this JVM.

System Class Loader

The system class loader locates its classes in those directories and JAR files specified in the `CLASSPATH` environment variable, or passed to it via the `-classpath` command-line option. The system class loader is also used to load an application's entry point class (that is, the class with the `main()` method), and is the default class loader for loading in any other classes not covered by the other two class loaders.

The system class loader is sometimes called the *application class loader*. This is a more intuitive term, as this class loader is loading up application classes. The IBM JVM uses this terminology; however, in this chapter we use Sun's convention, and call it a system class loader instead.

The Delegation Model

As discussed, Java SE has three different class loaders. If a `java.lang.String` is instantiated, the bootstrap class loader is responsible for loading its class, and if a user class (i.e., a class created by the developer) is instantiated, the system class loader is usually the responsible class loader. How does the JVM know which class loader to use?

The JVM knows which class loader to use by utilizing the *delegation model*. In every version of Java since JDK 1.2, whenever a class loader receives a request to load a class, it first asks its parent to fulfill the request (in other words, it *delegates* the request to its parent class loader). Before the class loader's parent loads the requested class, it delegates the request to *its* parent, and so on, until the bootstrap class loader is reached. If the parent is successful in loading the class, then the resulting class object is returned so that it may be instantiated (or statically referenced). Only if a class loader's parent (and its parent, and so on) fails to load the class does the original class loader attempt to load the class.

Thus, when a class is referenced in a Java program, the JVM will automatically route a request to the system class loader to load the necessary class. The system class loader then requests that the extension class loader load the specified class, which in turn requests that the bootstrap class loader load the class. The process stops with the bootstrap class loader, which then checks the core Java libraries (and whatever else it's configured to search) for the requested class.

If the class doesn't exist in the bootstrap class loader's domain, then the extension class loader will check the standard extensions location for the class. If it's still not found, then the system class loader will check the locations specified by the `CLASSPATH` variable for the class. If the class still cannot be located, then a `ClassNotFoundException` will be thrown. Figure 9-2 summarizes this process.

Consider the following example:

```
import com.wrox.MyObject;
public class Simple {
  public static void main(String[] args) {
    String myString = "test";
    MyObject myObject = new MyObject();
  }
}
```

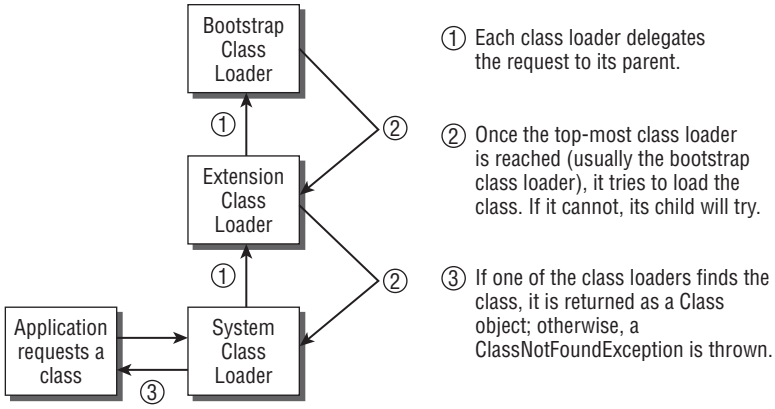
Bootstrap Class Loader
Extension Class Loader
System Class Loader
Application requests a class
① Each class loader delegates the request to its parent.
② Once the top-most class loader is reached (usually the bootstrap class loader), it tries to load the class. If it cannot, its child will try.
③ If one of the class loaders finds the class, it is returned as a Class object; otherwise, a ClassNotFoundException is thrown.

Figure 9-2: Delegation model

In this example, when the JVM sees the reference to `java.lang.String`, it requests that the system class loader try to load the class. However, before attempting to load the class itself, the system class loader requests that the extension class loader load it, and the extension class loader passes the request to the bootstrap class loader. Because the bootstrap class loader has no parent, it checks its paths for `java.lang.String` and finds it in the `rt.jar` file (at least, on Sun JVMs). The bootstrap class loader then returns the class down the chain until it is returned to the JVM.

The reference to `com.wrox.MyObject` also triggers the JVM to make the same request to the system class loader and, as with the `String` class, the system class loader delegates this to the extension class loader, which delegates to the bootstrap class loader. However, the bootstrap class loader won't find the `com.wrox.MyObject` class and returns nothing to the extension class loader. The extension class loader checks its paths, and it also won't find the class (unless, of course, the class has been explicitly placed in the extensions directory), and it, too, returns nothing to the system class loader. The system class loader searches in the `CLASSPATH` locations, where it finds the `com.wrox.MyObject`. The system class loader then returns the class to the JVM.

Endorsed Standard Override Mechanism

Java 1.4 introduced an interesting concept called the *Endorsed Standards Override Mechanism*, which is available in Java SE 5, too. Over time, the core Java SE distributions have been including more and more extensions (that at one time would have been considered optional) to Java. A good example of this was the inclusion of Java API for XML Processing (JAXP) with J2SE 1.4. Sun included both the JAXP API classes as well as an implementation of the API. Because the implementation classes are included in `rt.jar`, the bootstrap class loader loads them. As a result, if developers try to place newer versions of the JAXP implementations that shipped with the JDK in their `CLASSPATH`, their version will never be used. The system class loader will delegate all requests to the bootstrap class loader.

The problem is solved by the override mechanism. If developers place JAR files containing an alternate JAXP implementation in this class loader's domain, the bootstrap class loader will load their class files instead. In the Java SE 5, this location is as follows:

```
$JAVA_HOME/lib/endorsed
```

Users can change the path for this mechanism by setting the `java.endorsed.dirs` property, as shown next. In fact, the Tomcat startup script has such a setting for changing the path of the endorsed directory to `<CATALINA_HOME>/endorsed`:

```
$ java -Djava.endorsed.dir=/path/to/lib/endorsed ...
```

Before developers start thinking about replacing all of the core libraries, an important limitation must be addressed: Only certain packages can be overridden. The complete list of packages is in the Java SE 1.5 documentation (`http://java.sun.com/j2se/1.5.0/docs/guide/standards/`). In short, only the CORBA classes and the JAXP classes can be overridden with this mechanism.

More on Class Loader Behavior

Now that we've discussed the standard Java SE class loaders, as well as the delegation model that governs how these class loaders interact, the following sections address additional aspects of class loader behavior.

Lazy Loading (Loading Classes on Demand)

None of the three class loaders preloads all classes in the paths that they search for classes. Instead, they load the classes on demand. Such behavior is said to be *lazy* because the object waits to load the data until it is requested. While laziness in human beings is generally regarded as a negative trait, it is actually quite a positive one for class loaders, for the following reasons:

- **Faster performance:** At the time of initialization, if each class loader had to load every class, it would take much longer to initialize the JVM.
- **Efficiency:** Loading on demand results in more efficient memory usage because loading all the classes immediately would consume more memory than necessary.
- **Flexibility:** JAR files and classes can be added to the search paths of all the class loaders even after the class loaders have been initialized.

Note that when a class is loaded, all of its parent classes must also be loaded. Thus, if ClassB extends ClassA, and ClassB is loaded, then ClassA is also loaded.

Class Caching

The standard Java SE class loaders look up classes on demand, but once a class is loaded into a class loader, it will stay loaded (cached) for a period of time. However, the JVM's garbage collector can reclaim these `Class` objects. This is generally desirable, unless one such garbage-collected `Class` object is actually a stateful singleton class. (That is, a class that maintains a static reference to itself is either non-instantiable or not instantiated in practice, and maintains some aspects of an application's state.) For this reason, Sun JVMs allow class garbage collection to be turned off with the `-Xnoclassgc` option.

Separate Namespaces

Each class loader is assigned a unique namespace. In other words, if the bootstrap class loader loads a class named `sun.misc.ClassA`, and the system class loader loads a class named `sun.misc.ClassB`, the

two classes are considered to be in distinct packages. They do not have access to each other's package-private members.

Creating a Custom Class Loader

Java allows for the creation of custom class loaders, which may seem like one of those pointless tasks that only hard-core professional Java academics would ever want to perform. However, not only is creating custom class loaders fairly easy, but it can provide enormous flexibility in controlling aspects of an application's behavior.

The key to creating a custom class loader is the `java.lang.ClassLoader` class. This abstract class contains all the logic necessary for transforming the bytes of a compiled class file into a `Class` object that can then be used in an application. It does not, however, provide any mechanism for locating and loading such files.

The Java SE comes with two concrete implementations of `ClassLoader: SecureClassLoader` and `URLClassLoader`. The `SecureClassLoader` is a relatively thin wrapper around `ClassLoader` that ties class loading into Java's security model (security issues are discussed in the section "Security and Class Loaders," later in this chapter). Like `ClassLoader`, it does not provide a mechanism for loading class files.

`URLClassLoader` (a subclass of `SecureClassLoader`) provides the default Java mechanism for locating class files in directories or JAR files on a file system, or across a network. The extension and system class loaders are both descended from `URLClassLoader`, although they do not directly extend this class. Tomcat uses its own class loaders extensively. This is discussed in more detail later in this chapter.

Following are some neat tricks that you can perform with a custom class loader:

- Search a database instead of a file system for classes.
- Load different classes with the same fully qualified name.
- Swap your classes with new versions at runtime.
- Load classes before you need them.

Covering all the details associated with writing custom class loaders is an advanced development topic and beyond the scope of this chapter. More information about this topic can be gleaned from the `ClassLoader API` JavaDocs (`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html`).

Why Is a Custom Class Loader Needed for Tomcat?

A natural question to ask is why Tomcat (or any other Servlet container) needs custom class loaders? Custom class loaders allow Tomcat to do things like:

- Enforce rules for not loading application code from outside the `WEB-INF/classes` and `WEB-INF/lib` directory. The exception to this of course is the application code loaded by the "common" class loader, which is discussed later.
- Cache loaded classes for performance, instead of loading them every time they are requested.

- Preload classes, as specified. This allows Web applications to execute program logic at startup time, as well as in better performance.
- Reload classes at runtime, based on the timestamp on the class file. This feature allows for flexibility in application development, as it cuts down on the compile-deploy-test cycle. On production systems, this feature is often turned off, as it adds the overhead of having Tomcat check for the timestamp of Web application classes.

Security and Class Loaders

Class loading is at the very center of the Java security model. After all, it would clearly be undesirable for a rogue third party to be able to inject into an application a custom version of `java.lang.String` that had the nasty side effect of deleting the hard drive whenever it was instantiated. Understanding the security features of the class loader architecture will help you understand how Tomcat's class loader system works.

The Java class loader architecture tackles the security problem with the following strategies:

- Class loader delegation
- Core class restriction
- Separate class loader namespaces
- A SecurityManager

The following sections describe each of these strategies.

Class Loader Delegation

Recall the class loader delegation model discussed previously. Each class loader first determines whether its parent has the requested class before it attempts to load it.

The delegation model is described by many as a security feature. After all, it seems like it should be. Anyone trying to load false versions of the core Java classes will fail because the bootstrap class loader has first shot at any class, and it will always find the real copies of the core Java classes.

However, the delegation model on its own is flawed as a security mechanism because class loaders are *not required* to implement it. In other words, developers are free to create a class loader that doesn't follow the delegation model.

This security flaw doesn't impeach the class loader delegation model. Indeed, optional enforcement of the delegation model is actually an important feature, and other aspects of the class loader architecture resolve the security issues discussed here, as you will see shortly.

Core Class Restriction

If a custom class loader doesn't have to delegate requests to the system class loader, what would prevent it from loading in its own copy of `java.lang.String`?

Fortunately, it's not possible for any class loader written in Java to instantiate a core Java class. The `ClassLoader` abstract class (from which all class loaders must descend) blocks the creation of any class whose fully qualified name begins with `java`. Thus, no false `java.*` classes can be caught hanging around. Because the bootstrap class loader is not written in Java and does not descend from `ClassLoader`, it is not itself subject to this restriction.

By implication, this restriction indicates that all class loaders must at least delegate to the bootstrap class loader. Otherwise, when the class is loaded, there is no way for its class loader to load `java.lang.Object`, from which all objects must descend.

Thus, the delegation model by itself does not provide security. Instead, the core class restriction mechanism prevents rogue class loaders from tampering with the core Java libraries (at runtime).

Separate Class Loader Namespaces

As discussed previously, each class loader has its own namespace (thus, two different classes with the same fully qualified name). Because every single class loader has its own completely distinct namespace, class loader A can load a class named `com.wrox.Book`, and class loader B can also load a completely different class, also named `com.wrox.Book`.

Having separate namespaces is an important security feature because it prevents custom class loaders from stepping over each other, or the system class loader. No matter how hard a renegade class loader may try, it cannot replace a class loaded by a different class loader. Furthermore, classes loaded by different class loaders but otherwise in the same package cannot access each other's package-private members.

SecurityManager

If you really want to ensure that no damage can be done to a program with custom class loaders, you can disallow their use completely in an application. This can be done through the `SecurityManager` class, which is Java's general mechanism for applying security restrictions in applications.

With a SecurityManager and its associated policy files, you can disallow (or allow) a large number of tasks. For example, a program can be prevented from opening a socket to some network host, or be prevented from opening files on the local file system. In addition, of course, an application can be prevented from loading a class loader. Developers have the following options for preventing class loader–related operations:

- Prevent the loading of any class loader.
- Prevent a reference to any class loader being obtained (including the system class loader).
- Prevent the context class loader of any thread being changed.

Only two steps are required:

1. Configure a policy file with the permissions you want for a given application.
2. Turn on the application's SecurityManager.

For more detailed information on SecurityManagers, see Chapter 14.

Tomcat and Class Loaders

Recall the default Java class loader hierarchy, as summarized in Figure 9-3. Tomcat 6 builds on these class loaders by adding its own after the system class loader, as shown in Figure 9-4.

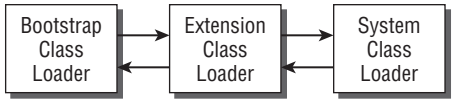

Figure 9-3: The Java class hierarchy

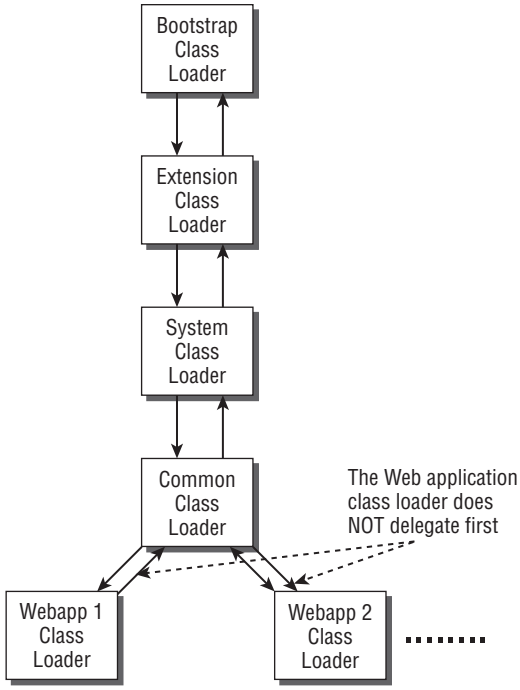
Bootstrap Class Loader
Extension Class Loader
System Class Loader
Common Class Loader
The Web application class loader does NOT delegate first
Webapp 1 Class Loader
Webapp 2 Class Loader

Figure 9-4: Tomcat's class loaders

If you are familiar with older versions of Tomcat, you will notice that there have been significant changes here: The older Catalina class loader and Shared class loaders have been dropped. Each of these additional Tomcat-specific class loaders is discussed in the following sections.

The Shared class loader can be added back by editing the `<CATALINA_HOME>/conf/catalina.properties` *file, and adding the following to the* `shared.loader` *property:*

```
shared.loader=${catalina.home}/shared,${catalina.home}/shared/*.jar
```

You can similarly add back the Catalina class loader by editing the `server.loader` *property, but there should never be a need to do that. Besides, this require splitting up the jar files under* `<CATALINA_HOME>/lib` *to make it work, and it is not worth the effort.*

System Class Loader

Tomcat uses the default system class loader, but it does something a little different from the default behavior of the JVM. In the Tomcat startup file (`startup.bat` on Win32, `startup.sh` on Linux, both of which in turn call `catalina.bat/sh`), the `CLASSPATH` environment variable is cleared. In its place, Tomcat points the `CLASSPATH` to two Tomcat files: `<TOMCAT_HOME>/bin/bootstrap.jar` and `<TOMCAT_HOME>/bin/tomcat-juli.jar`.

Recall that the system class loader searches the `CLASSPATH`. Because Tomcat sets the `CLASSPATH` variable to these two files, the normal effect of this class loader is nullified. Whatever the `CLASSPATH` is set to prior to launching Tomcat is simply disregarded by Tomcat.

The `bootstrap.jar` file contains Tomcat startup classes; `tomcat-juli.jar` contains the logging API implementation.

Endorsed Standards Override Mechanism

On startup, Tomcat changes the Endorsed Standards Override Mechanism to point to the following directory, rather than the default ones mentioned previously:

```
$CATALINA_HOME/ endorsed
```

Tomcat ships with a version of the popular Apache Xerces XML parser in this directory and a version of the JAXP API. The net result is that this parser (a JAXP implementation) is preferred to any that may have shipped with the JRE that was used to launch Tomcat (such as the Crimson parser that ships with Java 1.4).

Common Class Loader

Next in the hierarchy is the *common class loader*, which is responsible for classes that are used by Tomcat and publicly available to all Web applications. It loads all class files, resource files, and JARs located under the `CATALINA_HOME/lib` directory. This is a change from previous versions of Tomcat where the common class loader is used to load files from `$CATALINA_HOME/common/lib` and `$CATALINA_HOME/common/classes`.

Tomcat includes a number of JAR files in `$CATALINA_HOME/ lib`, such as Jasper (a JSP compiler), as well as the API classes for those APIs that Tomcat supports (Servlet, JSP, JNDI, and JMX).

Tomcat can reference all the classes included in the domain of this class loader in their own Web applications, and exclude such classes from their own Web applications (indeed, developers are not allowed to include the Servlet/JSP API classes in their Web applications). However, it's probably a good idea not to reference anything except the API classes, for the following reasons:

- Relying on the common class loader to load Jakarta Commons, Ant classes, and so on, from this directory potentially breaks Web application portability. No requirement is made of Servlet containers to provide such classes, and other Servlet containers probably don't provide them. Thus, when a Web application is moved from Tomcat to another such Servlet container, problems occur.
- The versions of such libraries that Tomcat includes may be different than the versions a Web application expects. Such bugs can be maddeningly difficult to track down.

Developers should not place their own classes or JARs in this class loader's domain. Another class loader, the *Web application class loader*, is provided for just this purpose and is discussed in the section "Web Application Class Loader," later in this chapter. Putting your own custom classes in the common class loader paths would be bad for at least two reasons, one trivial and one nontrivial:

- The trivial reason is that it's easy to forget which classes/JAR files are custom and which belong to Tomcat. Maintenance would, therefore, be tricky, especially for others who would not expect user classes to be in those locations.
- The nontrivial reason is that placing custom classes in the common class loader domain could cause compatibility problems with Tomcat or with other Web applications. For example, if an earlier version of the Xerces XML parser was placed in the domain, and it wasn't tested with Tomcat, it could cause mysterious bugs. The same would be true if an older version of the Servlet API were placed into these paths.

Web Application Class Loader

Each Web application also has its own class loader, which looks in the following locations:

- `$CATALINA_HOME/webapps/<webapp>/WEB-INF/classes`
- `$CATALINA_HOME/webapps/<webapp>/WEB-INF/lib`

Two properties of the *Web application class loader* make it unique. First, the Web application class loader does *not* use the delegation model that class loaders are encouraged to use. Instead, it tries to load classes first, before delegating the request to the other class loaders (except in certain conditions detailed later in this chapter). There are exceptions to this, however: The core Java classes (`java.*`, `javax.*`) cannot be overridden by the Web application class loader.

Second, each Web application has its own instance of this class loader, which means that no two Web applications can see each other's class files.

Web Application Class Loader Details

Actually, the Web application class loader does delegate to other class loaders; however, it does so in a way that is not consistent with the traditional delegation model.

When a class is requested, this class loader first checks its cache of classes to determine whether the class has already been loaded. If the class is not found among those cached, the Web application class loader will then delegate the request to the system class loader. This is done to prevent Web applications from attempting to instantiate classes shipped with the JRE.

If the system class loader fails to find the class, the Web application class loader will next try to determine whether the class belongs to any of the following packages:

```
javax.*
org.xml.sax.*
org.w3c.dom.*
org.apache.commons.logging.*
org.apache.xerces.*
org.apache.xalan.*
```

If the class does belong to one of these packages, the Web application class loader delegates the request to its parent, the common class loader.

If the class has still not been found, the Web application class loader will check its domain for the class. If it fails to find it, and it has not already delegated the request to its parent (that is, if the class belongs to one of the packages listed previously), it will do so now.

Class Loader Order Revisited

To review how these various Tomcat class loaders work together, let's examine what happens when an individual application requests a class. Following is a list of class loaders, and the order in which they will look for classes:

- The Web application class loader looks in `$CATALINA_HOME/webapp/<webapp>/WEB-INF/lib` and `$CATALINA_HOME/webapp/<webapp>/WEB-INF/classes` (except for those situations mentioned in the section "Web Application Class Loader Details," earlier in this chapter).
- The bootstrap class loader looks in the core Java classes.
- Under Java SE 5, the bootstrap class loader will also look in `$CATALINA_HOME/ endorsed` for alternative CORBA and JAXP classes.
- The system class loader looks in `$CATALINA_HOME/bin/bootstrap.jar` and `$JAVA_HOME/lib/tomcat-juli.jar`.
- The common class loader looks in `$CATALINA_HOME/lib`.

Dynamic Class Reloading

As discussed earlier, once a class loader has loaded a class, it caches the class. This means that future requests for the class always get the cached copy returned to them. Thus, if the class in the file system is changed while the JVM is running, the changed copy will be ignored.

However, because Tomcat uses its own class loader to load each Web application, it can accomplish dynamic class reloading simply by halting the Web application and then reloading it using a new class loader.

The Web application's original class loader is then orphaned and thus garbage-collected at the JVM's convenience. This eliminates the need to restart the JVM when new versions of classes are deployed.

Following are two mechanisms for instructing Tomcat to reload a Web application:

- Configure Tomcat to scan the Web application's `WEB-INF/lib` and `WEB-INF/classes` directories for changed files.
- Explicitly reload the Web application with the Tomcat `manager` application.

Note that in both cases, Tomcat does not simply direct its class loaders to dump their caches and reload from disk. Rather, when it detects a change or receives an explicit reload instruction, it reloads and restarts the entire Web application.

Because Tomcat cannot modify the JRE's built-in class loaders, classes loaded from their domains cannot be part of Tomcat's reload mechanism (that is, the contents of `$CATALINA_HOME/endorsed` won't be reloaded with this mechanism).

Performing either of these tasks is fairly simple and is described in Chapter 5.

Common Class Loader Pitfalls

A couple of common problems may occur when dealing with Tomcat's class loaders. The solutions to these problems, outlined in the following sections, are derived from information covered in the preceding sections.

Packages Split Among Different Class Loaders

If an application has multiple classes in the same package (for example, `com.wrox.servlets`), they must be loaded by the same class loader. For example, consider the following two classes:

```
com.wrox.servlets.MyServlet
com.wrox.servlets.Constants
```

These classes must be placed in the same class loader domain (such as `/WEB-INF/classes` or `$CATALINA_HOME/classes`). If they are split up, they no longer have access to each other's private, protected, or package-private (default) members.

Singletons

A *singleton* is a class designed so that it can be instantiated only one time in any given JVM. Consider the example of a singleton class that is an entry point to an object pool of some sort. The designers of this class want to share this singleton among multiple Web applications, and want to maintain the contract that only one instance be created in a single JVM.

The singleton class could look something like the following:

```
public class ObjectPool {
    private static ObjectPool objectPool = null;
    private ObjectPool {
        // initialize object
    }

    public synchronized static ObjectPool getInstance() {
        if (objectPool == null) {
            objectPool = new ObjectPool();
        }
        return objectPool;
    }
}
```

Placing this class in the Web application class loader domain guarantees that each Web application will create a new instance of this class. This is because each Web application has its own class loader, and class loaders maintain distinct namespaces.

The solution is to place this class in the common class loader domain, where the singleton will be shared among all Web applications because they all share the same class loader.

Recall, however, that the JVM can garbage-collect loaded class objects if memory is low. If the singleton is not currently referenced by any classes in the JVM, it could be garbage-collected, even when its contents are still important to the application. This scenario is a regrettable side effect of the singleton design pattern. Because they can be accessed statically at any time, applications need not maintain a reference to them. Without a reference, the JVM garbage collector cannot determine that the class is currently in use by the application.

Sun's JVMs provide a solution to this problem. Use the `-Xnoclassgc` startup parameter. This parameter can be utilized when launching Tomcat by setting the `JAVA_OPTS` environment variable to this value. The following is an example of doing this on Windows with the Sun JVM:

```
set JAVA_OPTS=-Xnoclassgc
```

The corresponding option in IBM's JVM is `-noclassgc`. Turning off garbage collection is not advisable in many cases, especially if you have a long running application that frequently loads classes.

Another solution is to ensure that the singleton class is always referenced in some way by the application. This can be as simple as adding the following member to a class that is always loaded in an application:

```
private ObjectPool objectPool = ObjectPool.getInstance();
```

XML Parsers

Unfortunately, the whole issue of XML parsers in Java has become somewhat confusing. Java defines an API for XML parsers based on the W3C organization's DOM standard, called the Java API for XML Processing, or JAXP.

Starting with Java 1.4, the JAXP API was included in the Java SE platform, along with an implementation of JAXP: the Apache Crimson parser. However, Crimson was soon discontinued by the Apache group, in favor of the Xerces XML parser. Xerces was also soon discontinued, in favor of Xerces2.

As a result of this high level of churn, the Java community must now deal with having multiple JAXP implementations in the marketplace and *many* versions of those parsers. Because these disparate versions often have classes with the exact same fully qualified names, using JAXP can sometimes lead to weird results and buggy behavior. The situation is further complicated by the presence of multiple versions of JAXP.

As mentioned previously, the Endorsed Standards Override Mechanism exists to prevent the JRE's class loaders from loading the JAXP classes in situations where an alternate JAXP API version or alternate implementation is desired. Tomcat uses this mechanism to sidestep the XML parser issues by including its own JAXP API and implementation (Xerces).

Unfortunately, under Java 1.4, Tomcat must share the same version of the Xerces XML parser as its Web applications (as well as the same version of Xalan, an XSLT engine). Tomcat relies on the *Endorsed Standards Override Mechanism* for the version of Xerces it requires, and as discussed previously, the Web application class loader delegates to this mechanism under all conditions (see the section "Web Application Class Loader Details," earlier in this chapter).

This is likely not a problem unless a Web application relies on an old version of Xerces/Xalan for a particular behavior. Or, it is possible that a version of Xerces/Xalan that is newer than the one that shipped with Tomcat is required for a Web application (the version of Xerces/Xalan shipped with Tomcat can be determined by checking the `MANIFEST.MF` file in the Xerces JAR). In such circumstances, the only alternative is to change the parser in the `$CATALINA_HOME/endorsed` directory and hope that such a change doesn't break Tomcat.

Of course, these parser version issues really apply only if the Apache family of XML parsers is used. If you use another JAXP implementation whose classes are in entirely different package names, all of the issues just described are eliminated. However, Xerces is by far the world's most popular JAXP implementation.

Summary

To conclude this chapter on class loaders, let's review some of the key points we have discussed:

- Class loaders abstract the process of loading class files before the first instantiation and make them available for use. Java's default class loaders support loading classes from the local file system and from a network. Java also provides developers with a facility to create their own custom class loaders. The three basic class loaders discussed are the bootstrap, extension, and system class loaders.
- Class loaders use the delegation model. Every class loader passes a request to load a class to its parent until the bootstrap class loader is reached. Each class loader looks for the class, and if it can't be found, the request goes back down the chain. Implementing the delegation model is optional, but class loaders are basically useless if they don't delegate to the bootstrap class loader at some point. Every class loader has a unique namespace.
- The Java security model prevents the misuse of custom class loaders by allowing only the bootstrap class loader to load classes that start with `java.*`. By using the SecurityManager, an application can forbid the use of custom class loaders.
- Tomcat has two different class loaders: common, and Web application. This is in addition to the standard Java classloaders — bootstrap, extension, and system.
- Classes and resource files specific to a Web application should be placed under the Web application's `WEB-INF/classes` directory. Similarly, JAR files specific to a Web application should be placed under the Web application's `WEB-INF/lib` directory.

Chapter 10 examines the first of the Tomcat Connectors: HTTP Connectors.

10

HTTP Connectors

When used out of the box to run Web applications, Tomcat can serve HTML pages without any additional configuration. This works because Tomcat has been preconfigured with an HTTP Connector that can handle requests from a user's Web browser. Because of this Connector, Tomcat can function as a standalone Web server. It can serve static HTML pages, as well as handle servlets and JSP pages.

Tomcat Connectors provide the external interface (over HTTP or HTTPS) to Tomcat clients. There are two kinds of Connectors — those that implement an HTTP stack of their own (called *HTTP Connectors*) and those that tie Tomcat to an external Web server such as Apache or IIS (called *Web server Connectors*). This chapter examines in detail the configuration of the various HTTP Connectors available in Tomcat 6. Chapters 11 and 12 discuss Web server Connectors.

The Java-based HTTP/1.1 Connector is the default Connector configured for Tomcat 6. It is an evolved version of the Java-based HTTP/1.0 and HTTP/1.1 Connectors appearing earlier in Tomcat 5.*x*.*x* versions. An earlier version of this mature Java-based connector is called the Coyote Connector, and has been available as an add-on from as far back as Tomcat 3.*x*.

In addition to the default configured Java-based HTTP Connector, Tomcat 6 provides a variety of alternative HTTP Connectors. A Java HTTP Connector written to take advantage of the high performance IO features of the Java 5 NIO library is available. In addition, a native version of the HTTP Connector, written in C/C++ and coded to APR (Apache Portable Runtime) is also available. While both of these alternative Connectors are relatively new compared to their default Java Connector cousin, they hold much promise for the near future.

Even though very little additional configuration is required to get the HTTP Connector working, you may want to fine-tune some of its features. This chapter describes what to do when your Connector configuration needs to be modified, such as for specific deployments (for example, running Tomcat behind a proxy), SSL setup, or performance tuning.

The following areas are covered in this chapter:

- Using Tomcat 6 default Java HTTP/1.1 Connector
- Using Tomcat 6 Java NIO (Advanced IO) HTTP/1.1 Connector
- Using Tomcat 6 native APR (native code) HTTP/1.1 Connector
- Blocking and non-blocking operations
- Supporting Comet
- Running Tomcat behind a proxy server
- Setting up SSL
- Performance tuning

HTTP Connectors

The standard HTTP Connectors included with Tomcat 6 provide the ability to run Tomcat in a standalone mode. In this mode, Tomcat can respond to HTTP requests directly from users' browsers — without the assistance of a separate Web server. In addition to requests for servlets and JSPs, the HTTP connectors also respond to requests for static contents, such as static Web pages and images.

The HTTP Connectors are Java classes that implement the HTTP protocol. An HTTP connector is invoked when there is an HTTP request on the Connector port. The port that the Connector listens on is specified in the `$CATALINA_HOME/conf/server.xml` configuration file, and is set to 8080 by default. The Connector class has code to parse the HTTP request and take the required action of either serving up static content or passing the request through the Tomcat Servlet Engine. The HTTP Connector implements the HTTP/1.1 protocol, and all protocol features. For clients that support only HTTP/1.0, the Connector degrades gracefully to support the legacy protocol.

There are multiple HTTP Connector implementations available with Tomcat 6. These include:

- Java-based HTTP/1.1 (Coyote) Connector
- Java-based High Performance NIO HTTP Connector
- Native code–optimized APR HTTP Connector

The Java-based Coyote Connector is the most mature of the three variations, it is extremely stable, works well, has great performance, and should be the choice for most situations — especially if you are just beginning to use Tomcat 6.

The high-performance NIO HTTP connector provides non-blocking IO and Comet support, if you are using a software library that requires (or benefits from) this additional support, you may want to use this connector instead. Be aware, however, even as late as Java SE 5, there are serious bugs in the Java NIO library that may affect stability of your Tomcat server.

On highly loaded systems, the optimization possibilities offered by the APR HTTP Connector can be very attractive. This is the newest of the three connector implementations, but potentially offers the highest level of performance improvement by leveraging proven operating system–level optimization on

a native code interface level. After this connector has been fully field tested by the production Tomcat 6 user community, it is likely to become the Connector of choice on platforms that are supported by the Connector (currently including Win32 and Linux platforms).

This chapter explores the properties and configuration of each implementation in more detail.

Tomcat 6 HTTP/1.1 Connector

The standard Java-based HTTP/1.1 Connector in the Tomcat 6 server is the Coyote Connector. This Connector is the most mature and stable of all the available Tomcat 6 HTTP connectors.

HTTP/1.1 Connector Configuration

A typical standard HTTP/1.1 Connector configuration is as follows (taken from `$CATALINA_HOME/conf/server.xml`):

```
<Connector port="8080"
           protocol="HTTP/1.1"
           maxThreads="150"
           connectionTimeout="20000"
           redirectPort="8443" />
```

Although the only mandatory attribute for the Connector configuration is the `port` attribute, numerous other important attributes can be configured, as described in the following list:

- `acceptCount`: This is the maximum queue length for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused. This value is passed as the backlog parameter while creating a Tomcat server socket. The default queue length is 10, and the maximum is operating system-dependent.
- `address`: This attribute specifies the IP address to which the Tomcat server binds. If the `address` attribute is not specified, Tomcat would bind to all addresses (if the host has multiple IP addresses).
- `allowTrace`: This enables the `TRACE HTTP` method if set to `true`. The default is `false`.
- `compressibleMimeTypes`: This is a comma-separated list of MIME types for which HTTP compressions (see the next attribute) can be used. The default value is `text/html,text/xml, text/plain`.
- `compression`: The Connector can use HTTP/1.1 GZIP compression to get better bandwidth from the server. This can be enabled via the `compression` attribute. The valid values are `off` (disables compression), `on` (enables compression), `force` (forces compression in all cases), or a numerical value that specifies the minimum amount of data required before the output is compressed. The default value of the compression attribute is `off`.
- `connectionLinger`: This sets the number of milliseconds for which socket connections will persist after the connection is closed. A value less than `0` means don't linger (this is the default).
- `connectionTimeout`: This is the number of milliseconds that this Connector waits after accepting a connection before requesting the URI line to be presented. The default value is 60,000 milliseconds (60 seconds).

- `disableUploadTimeout`: This attribute enables a separate timeout to be set (or not set) for data uploads during a servlet execution. The attribute's value defaults to `false`.
- `emptySessionPath`: Default is `false`. The session path used for cookies is all `"/"` if set to `true`. In general, set this to `true` only when the server is used to run applications written to the Portlets specification.
- `enableLookups`: When this is set to `true`, all calls to `request.getRemoteHost()` perform a DNS lookup to return the host name for the remote client. When this attribute is `false`, the DNS lookup is skipped and only the IP address is returned. The default value for `enableLookups` is `false`. Keeping this attribute turned off increases performance, which enables you to avoid the overhead required for the DNS lookup. These and other performance considerations are discussed in the section "Performance Tuning," later in this chapter.
- `maxHttpHeaderSize`: This attribute controls the maximum size of the request and response headers. The unit is bytes. The default value is 4096 (4K).
- `maxKeepAliveRequest`: This attribute controls the "keep-alive" behavior of HTTP requests that enables persistent connections (that is, multiple requests to be sent over the same HTTP connection). It specifies the maximum number of requests that can be pipelined until the connection is closed by the server. The default value of `maxKeepAliveRequest` is `100`, and setting it to `1` disables HTTP keep-alive behavior and pipelining.
- `maxPostSize`: This specifies the maximum size, in bytes, of the `POST` that can be handled by the container. It defaults to 2,097,152 (2MB). If set to `0` or a negative number, this feature is disabled.
- `maxSavePostSize`: This specifies the maximum size, in bytes, of the `POST` that can be handled by the container during a client-cert or form authentication operation. It defaults to 4096 (4K). If set to –1, this feature is disabled and POSTed data is not saved during the two types of authentication.
- `maxSpareThreads`: The `maxSpareThreads` attribute controls the maximum number of unused threads that are allowed to exist before Tomcat starts stopping the unused ones. `maxSpareThreads` defaults to 50.
- `minSpareThreads`: The `minSpareThreads` attribute specifies the minimum number of threads that are started when the Connector is initialized. `minSpareThreads` defaults to 4.
- `maxThreads`: This attribute specifies the maximum number of threads that are created for this Connector to process requests. This, in turn, specifies the maximum number of concurrent requests that the Connector can handle. `maxThreads` defaults to 200 threads.
- `noCompressionUserAgents`: This is a comma-separated list that matches the HTTP UserAgent value of Web browsers that have a broken support for HTTP/1.1 compression. Regular expressions can be used here.
- `port`: The `port` attribute specifies the TCP port number on which this Connector will create a server socket and await incoming connections. Only one server application can bind to a particular port number–IP address combination.
- `protocol`: This specifies the HTTP protocol to use, and must be set to `HTTP/1.1` (the default). This loads the default `org.apache.coyote.http11.Http11Protocol`.
- `implementation`: This is the default Java-based blocking connector. If you specify `org.apache.coyote.http11.Http11NioProtocol`, the non-blocking NIO connector is selected. If you specify `org.apache.coyote.http11.Http11AprProtocol`, the APR connector is

selected. The APR connector is also used if the APR library is available via either the `PATH` variable in Windows or `LD_LIBRARY_PATH` variable in Linux/*nix.

- `proxyName`: The `proxyName` attribute (along with the `proxyPort` attribute) is used when Tomcat is run behind a proxy server. It specifies the server name to be returned for `request.getServerName()` calls. See the section "Running Tomcat Behind a Proxy Server" later in this chapter for more information.
- `proxyPort`: As mentioned, the `proxyPort` attribute is used in proxy configurations. It specifies the port number to be returned for `request.getServerPort()` calls. See the section "Running Tomcat Behind a Proxy Server" later in this chapter for more information.
- `redirectPort`: If the Connector supports only non-SSL requests and a user request is sent to this Connector for an SSL resource, Catalina will redirect that request to the `redirectPort` port number. The default Tomcat configuration specifies `8443` as the redirect port, as shown in the sample configuration presented earlier. If this is omitted, it defaults to `443`.
- `restrictedUserAgents`: This is a comma-separated list that matches the HTTP UserAgent value of Web browsers that have a broken support for HTTP/1.1 keep-alive behavior. Regular expressions can be used here.
- `scheme`: The `scheme` attribute is set to the name of the protocol. The value specified in `scheme` is returned by the `request.getScheme()` method call. The default value is `http`. For an SSL Connector, this would be set to `https`.
- `secure`: This attribute is set to `true` for an SSL Connector. This value is returned by the `request.getScheme()` method calls. The default value is `false`.
- `server`: Specifies the server header when sending the HTTP response. If this attribute is not set, the Server string output in the HTTP header that identifies the Web server defaults to `Apache-Coyote/1.1`. Some security experts don't like this, as it broadcasts to the world information about the Web server software. If there are any known security exploits of this Web server, they then can be used by malicious users. Setting this to an empty string suppresses the printing of the `Server` string.
- `socketBuffer`: This specifies the size, in bytes, of the buffer to be used for socket output buffering. Use of a socket buffer helps to improve performance. By default, a buffer of size 9,000 bytes is used, and setting `socketBuffer` to `-1` turns buffering off.
- `tcpNoDelay`: When this attribute is set to `true`, it enables the `TCP_NO_DELAY` network socket option. This improves performance, as explained in the section "Performance Tuning," later in this chapter. The default value is `true`.
- `threadPriority`: Specifies the Java thread priority for request handling threads created in the Java VM. The default value is `java.lang.Thread#NORM_PRIORITY`.
- `URIEncoding`: Specifies the character encoding used to decode URI bytes. It defaults to `ISO-8859-1`.
- `useBodyEncodingForURI`: If set to `true`, this attribute causes the URI encoding specified in the `contentType` to be used for encoding, rather than the `URIEncoding` attribute. This defaults to `false`.
- `useIPVHosts`: Default is `false`. If set to `true`, this attribute causes the server to examine the incoming request IP address to direct the request to the corresponding virtual host.

- `xpoweredBy`: If set to `true` (the default value is `false`), an `X-Powered-By` header is output in servlet-generated responses returned by the Connector. The value of the header is `Servlet/2.5`, as shown in the following sample HTTP response header:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Content-Type: text/html
Content-Length: 1437
Date: Thu, 07 Aug 2007 17:25:52 GMT
Server: Apache-Coyote/1.1
```

In addition to these attributes, there are others that are specific to SSL Connectors. These are valid only if the `secure` attribute is set to `true`, and are discussed next.

Configuring Tomcat 6 for SSL

The Connector for the Catalina instance that supports HTTPS connections must have its `secure` attribute set to `true` and its `scheme` attribute set to `https`. Unlike Tomcat 4.*x*, no Factory element is required for SSL-related configuration, although it is still supported for backward compatibility.

The new SSL-related Connector attributes are as follows:

- `algorithm`: Specifies the certificate encoding algorithm to use. This defaults to `SunX509`.
- `ciphers`: A comma-separated list of encryption ciphers.
- `clientAuth`: Can be set to either `true` or `false` (the default is `false`). When set to `true`, the client connection would need to present a valid certificate. However, if `clientAuth` is set to `false`, and the Web resource being requested is protected by `CLIENT-CERT` authentication, the latter would take precedence (that is, the client would still need to present a certificate).
- `keystoreFile`: Specifies the pathname to the keystore file. The keystore file contains the server's public and private keys in the form of certificates. `keystoreFile` defaults to `.keystore` in the user's home directory. The home directory varies by operating system (for example, `/home/`*`user_name`* in Linux; and `C:\Documents and Settings\`*`windows user name`* for Windows).
- `keystorePass`: Should be set to the password required to access the `keystoreFile`. The default password is `changeit`.
- `keystoreType`: Specifies the keystore file type. It defaults to `JKS` (Java Keystore). This format is SUN's format supported by the underlying Java SE platform implementation and need not be changed if you are setting up a new Java keystore. If your underlying operating system already has a keystore in another format, and you have installed a JCE provider that supports that format in your JDK, you may be able to use it. For example, "pkcs11" and "pkcs12" are supported by some JDK and operating system combinations.
- `sslProtocol`: This indicates which version of the SSL protocol to use (the default value is `TLS`). The default value is fine for most configurations, as TLS is essentially SSL3. For backward compatibility with some older JVM, if you are running on old Java VMs, you may need to change this to SSL.

Following is an example Connector with SSL configuration. This configuration is already present in the `$CATALINA_HOME/conf/server.xml` file, but is commented out. Note that if the SSL port (8443) is

changed, the `redirectPort` attribute for all the non-SSL Connectors must be changed to that port number, too. As mentioned, the non-SSL Connectors redirect users to this port if they try to access pages with a security constraint that specifies that SSL is required.

For more on installing an SSL certificate that you have purchased from a Certificate Authority, and/or creating your own self-signed certificate for testing, see the section "Encryption with SSL" in Chapter 14.

```
<Connector port="8443"
           protocol="HTTP/1.1"
           maxThreads="150"
           scheme="https"
           secure="true"
           clientAuth="false"
           sslProtocol="TLS" />
```

A different set of SSL configuration attributes is required when `OpenSSL` is used with the native code APR Connector. Please see the later section on configuration for the native APR connector.

Note that when you startup Tomcat and examine the output of the console, you are very likely to see the following message:

```
INFO: The Apache Tomcat Native library which allows optimal performance in
production environments was not found on the java.library.path: c:\jdk ...
```

This message appears because a listener is enabled by default to look for the APR support libraries; this message indicates that the APR libraries are not found. You need these libraries only if you are configuring the APR connector. See the "Native APR Connector" section in this chapter for more details.

The Advanced NIO Connector

The NIO Connector is an HTTP Connector written in Java, but takes advantage of the NIO libraries of Java 5 to provide the following features, which are above and beyond those provided by the standard HTTP Connector:

- Non-blocking operations
- Support for Comet

This Connector has the same configuration attributes as the standard HTTP Connector. It is activated when the protocol attribute is set to `org.apache.coyote.http11.Http11NioProtocol`.

For example, the following `<Connector>` element configures an instance of an NIO Connector:

```
<Connector port="8080"
           protocol="org.apache.coyote.http11.Http11NioProtocol"
           maxThreads="150"
           connectionTimeout="20000"
           redirectPort="8443" />
```

Comet Asynchronous IO Support

If Servlets need to take advantage of the non-blocking IO capabilities, the NIO connector must be explicitly coded for asynchronous IO. This asynchronous IO support is called Comet.

Using Comet, a Servlet can respond to incoming events from a client instead of blocking in a read; it can also write responses asynchronously.

Comet Servlets implement the `org.apache.catalina.CometProcessor` interface, instead of the usual Servlet interface. Through this interface, the Servlet can receive a sequence of events. The following describes some of the events that a Comet Servlet can receive:

- `BEGIN`: The Servlet responds to this event by performing any initialization; this signifies the beginning of the request/response lifecycle; the response object will be available for writing back to the client, but access must be synchronized if multiple threads are involved in this stage.
- `READ`: An event that signifies to the Servlet that one read may be performed on the request stream without blocking; the Servlet must perform this read. Multiple `READ` events will be fired as request data becomes available.
- `END`: The Servlet is notified that the request processing should now end; the Servlet should complete the writing of the response and reset any initialized fields on the request or response objects.
- `ERROR`: This event from Tomcat 6 notifies the Servlet that an IO Exception or other fatal error has occurred during processing.

While servicing any of the preceding events, the Servlet can decide to end processing by calling the `close()` method on the event.

Because the benefit of asynchronous IO can be enjoyed only by Servlets that are explicitly coded to the preceding event-driven model, there is very little reason to configure the NIO Connector for Web applications consisting exclusively of regular servlets that do not support Comet.

The Native APR Connector

APR is the Apache Portable Runtime, which is a native code library that enables C/C++ software to be written in a platform-independent fashion. APR works well across Windows, Linux, and *nix systems. The Native APR Connector is written using APR, and compiled to native code for optimized platform-specific performance.

The Native APR Connector is not a complete Connector in the strictest sense. It actually makes use of the standard Java-based connector for most of its operations. However, when the native code APR connector is enabled, the Java code will switch to native implementation for several performance- and scalability-sensitive operations. The APR Connector optimizes performance and enhances scalability via three main mechanisms:

- Use of a `sendfile()` kernel mode call to send large static files directly from the buffer cache
- Use of a single native code keep alive poller to implement connection keep alive for a large number of connections
- Use of the `OpenSSL` native code, which has the potential to accelerate SSL implementation for SSL handling (via hardware)

Enabling the APR Connector

The APR Connector is configured under the following conditions:

- The protocol attribute is set to `org.apache.coyote.http11.Http11AprProtocol`.
- The APR library is located in the `PATH` environment variable (Windows) or the `LD_LIBRARY_PATH` (*nix/Linux).

If you do not have the APR library already installed for your platform (most Linux distributions come with the APR already installed), you may need to download a binary distribution first.

At the current time, the source and binaries for the Windows APR connector implementation (with OpenSSL) can be downloaded from the following URL:

```
http://tomcat.heanet.ie/native/
```

The location may change in the future. For Linux, the source is included with the Tomcat distribution; you can find it in the `bin/tomcat-native.tar.gz` file. See Chapter 3 for information on installing APR on Linux.

As an example, if you have a Win32 system (XP, Vista, and so on), download the binaries from the preceding URL. Place the two downloaded binary files — `tcnative-1.dll` and `openssl.exe` — into the `$CATALINA_HOME/bin` directory. For example, the following `<Connector>` element configures an instance of an APR Connector, provided the `PATH` or `LD_LIBRARY_PATH` environment variable contains a path to the APR libraries.

```
<Connector port="8080"
           protocol="org.apache.coyote.http11.Http11AprProtocol"
           maxThreads="150"
           connectionTimeout="20000"
           redirectPort="8443" />
```

Configurable Connector Attributes

The setting of configurable attributes on this Connector overlaps with the standard HTTP Connector. The following are attributes from the standard HTTP Connector that are also applicable to the native APR Connector. See the standard HTTP/1.1 Connector section, earlier in this chapter, for a description of these attributes.

- `allowTrace`
- `emptySessionPath`
- `enableLookups`
- `maxPostSize`
- `maxSavePostSize`
- `protocol`
- `proxyName`
- `proxyPort`

- `redirectPort`
- `SSLEnabled`
- `scheme`
- `secure`
- `URIEncoding`
- `useBodyEncodingForURI`
- `useIPVHosts`
- `xpoweredBy`

Other APR Connector–specific attributes are described in the sections to follow.

Kernel Mode File Transfer Sendfile() Optimization

One of the core features of enabling APR is the ability to send a large file via a kernel-level `sendfile()` system call. This call is optimized for sending large static files through a socket. Instead of repeatedly copying the file data to higher level buffers, such as those maintained in byte arrays within the Java VM, the kernel mode API takes care of sending the file directly from the file system's buffer cache. Although this `sendfile()` operation is performed synchronously by the kernel, it is asynchronous with respect to the Java VM. This theoretically enables the Tomcat server to perform other work while the file is being sent by the lower-level call. This feature cuts down on CPU time spent on data copying as well as minimizing the context switches between the Java VM and kernel mode operations during the sending of very large file.

On systems without the kernel mode `sendfile()` system call, the Tomcat Connector gracefully falls back on the Java-based buffer IO to send large static files. In addition, the `sendfile()` operation does not take effect when SSL is used with the connector.

The default Servlet, configured in the `conf/web.xml` file, is responsible for serving static content and has a `sendfileSize` attribute that you can configure to control the minimal size of a static file being considered for transmission via the `sendfile()` call.

Scalable Keep-Alive Poller

The keep-alive poller is an APR component responsible for maintaining keep-alive connections. The number of kernel modes to the Java VM context switches is reduced when a native code component is used to keep track of keep-alive connections.

The following additional attributes are available for configuration with the APR Connector enhancements:

- `firstReadTimeout`: This value, in milliseconds, controls the timeout set on a connection's first read call. If this timeout is reached, the connection is handed off to the keep-alive poller. Note that if this timeout value is never reached, the keep-alive poller does not get involved with the connection. If you want to use the keep-alive poller every time, you can set the value to `0`, `-1`. Both will hand off connections to the poller on every read; however, `0` tells the poller to use a very short timeout, and `-1` indicates the use of the connection's configured socket timeout value. Default value is `-1`. Note that `-2` can also be used to request the runtime to bypass the use of the poller as much as possible.

- `pollTime`: Adjusts the timing between poll calls. The more frequently these calls are made, the faster the managed sockets will respond to keep alive termination. However, this also results in increased CPU load because of the more frequent polling. The default value specifies the number of poll calls made per second; the default is 2000 (polling 5 ms apart). This means that in the worse case, a socket destined for keep-alive termination may not be detected until 5 ms after its targeted disconnection time.
- `pollerSize`: This is used to control the maximum number of sockets available to the keep-alive poller. Default is 8192.
- `useSendfile`: The default value is `true`. This is a flag to indicate if the kernel-based `sendfile()` optimization should be used.
- `sendfileSize`: This is used to control the maximum number of sockets used by the poller for `sendfile()`-based operations. Default is `1024`. On platforms where the `sendfile()` call returns asynchronously, the poller is not used and the maximum number of concurrent static file transfers may not correspond to this value.

OpenSSL Support

`OpenSSL` is a popular open source SSL library written in optimized C/C++ code, and available on most operating systems, including Windows and *nix/Linux. `OpenSSL` is a very mature library of robust and high-performance code and is used by the industry-leading Apache Web server in its `mod_ssl` implementation.

Because of its popularity, there is a thriving secondary market for enhancements to the library. One of the most interesting areas is the availability of a hardware-accelerated SSL encryption solution that is compatible with `OpenSSL`. The native APR connector enables Tomcat 6 to take advantage of `OpenSSL` and all the optimized code and associated acceleration technology.

The following additional configuration attributes are available in the APR Connector for customization of `OpenSSL`:

- `SSLEnabled`: This controls whether the APR Connector will use `OpenSSL` instead of JSSE (Java Secured Socket Extension — a built-in library of Java SE 5) for encryption and decryption. Default is `false`, meaning that the JSSE implementation will be used.
- `SSLProtocol`: Selects the protocols that may be used by the `OpenSSL` implementation to communicate with clients. The default is `all` and there is little reason to change this. But if you want more precise restrictions on the protocol used, you may set it to `SSLv2`, `SSLv3`, `TLSv1`, or `SSLv2+SSLv3`.
- `SSLCipherSuite`: Selects or restricts the cipher that can be used by the `OpenSSL` implementation. The default is `all`; there is little reason to change this. The possible values are documented with `OpenSSL`; you can check `openssl.org/docs/apps/ciphers.html`.
- `SSLCertificateFile`: Specifies the file that contains the server-side certificate, encoded in PEM (Base64) format, used by `OpenSSL`. There is no default value for this property.
- `SSLCertificateKeyFile`: Specifies the file containing the PEM (Base64)–encoded server private key for `OpenSSL` operations. The default value is set to the same value as the `SSLCertifcateFile` property.

- `SSLPassword`: Specifies the passphrase used for the server private key. No default value. If not specified, the user is prompted for a password when the key is used.
- `SSLVerifyClient`: Sets client authentication. The default is `none`, meaning client authentication is not enabled. You can also set this to `optional`, `require`, or `optionalNoCA`.
- `SSLVerifyDepth`: Sets the maximum accepted verification depth for client certificates. The default is `10`.
- `SSLCACertificateFile`: Sets the file containing PEM (Base64)–encoded certificates of CAs for client authentication. This single file should contain the certificates of all the client authentication CAs concatenated. You can either use this or the `SSLCACertificatePath`, described next, to specify CA certificates for client authentication.
- `SSLCACertificatePath`: Sets the directory containing CA certificates for client authentications. Each individual file in this directory contains a single PEM (Base64)–encoded CA certificate used for client authentication. The filenames used for these files are `mod_ssl` generated hash symbolic links; these files are typically placed here using the makefile from `mod_ssl`.
- `SSLCertificateChainFile`: Sets the file containing the server CA certificates chain. This file contains a concatenation of the PEM (Base64)–encoded CA certificates in chain order.
- `SSLCARevocationFile`: Sets the file containing PEM (Base64)–encoded CA certification revocation lists for certificate revocation during client authentication. This single file should contain the Certification Revocation Lists of all client-authentication CRLs concatenated. You can either use this or the `SSLCARevocationPath`, described next, to specify CA certificates for client authentication.
- `SSLCARevocationPath`: Sets the directory containing CA Certificates Revocation Lists for certificate revocation during client authentication. Each individual file in this directory contains a single CA Certificates Revocation List used for certificate revocation. The filenames used for these files are `mod_ssl`–generated hash symbolic links; these files are typically placed here using the makefile from `mod_ssl`.

Configuring Tomcat for CGI Support

Support for CGI in Tomcat 6 is accomplished by a Servlet (the `org.apache.catalina.servlets.CGIServlet`) that simulates the way a Web server would handle a CGI script — processing the CGI environment variables and then executing the CGI executable.

However, CGI is disabled in the default Tomcat configuration for security reasons. For instance, a CGI script would bypass the security policies defined for programs in the `catalina.policy` file. More information about these security policies is provided in Chapter 14.

Enabling CGI support across all applications in a Tomcat server requires the following steps.

There are performance and security issues with CGI, and unless you have to support legacy application code, you should not enable it in a production environment. See Chapter 14 for more details.

1. Uncomment the `servlet` and `servlet-mapping` settings for CGI in `CATALINA_HOME/conf/web.xml` — these settings are commented by default. The `servlet-mapping` causes all requests for Web pages with a `/cgi-bin/` prefix to be passed to the CGI Servlet, and the `servlet` element specifies the fully qualified Java class name of the servlet and its configurable parameters.

The sample settings are shown in the following code and the configurable parameters are as follows:

- `cgiPathPrefix`: The directory containing CGI scripts
- `debug`: The debug level to be enabled for the CGI servlet
- `executable`: The program used to run CGI
- `parameterEncoding`: The text encoding used for parameters to the CGI scripts
- `passShellEnvironment`: Determines whether the shell environment is passed to the CGI scripts

```
<!-- Common Gateway Includes (CGI) processing servlet, which supports     -->
<!-- execution of external applications that conform to the CGI spec      -->
<!-- requirements.  Typically, this servlet is mapped to the URL pattern  -->
<!-- "/cgi-bin/*", which means that any CGI applications that are         -->
<!-- executed must be present within the web application. This servlet    -->
<!-- supports the following initialization parameters (default values     -->
<!-- are in square brackets):                                             -->
<!--                                                                      -->
<!--   cgiPathPrefix        The CGI search path will start at             -->
<!--                        webAppRootDir + File.separator + this prefix. -->
<!--                        [WEB-INF/cgi]                                 -->
<!--                                                                      -->
<!--   debug                Debugging detail level for messages logged    -->
<!--                        by this servlet.  [0]                         -->
<!--                                                                      -->
<!--   executable           Name of the exectuable used to run the        -->
<!--                        script. [perl]                                -->
<!--                                                                      -->
<!--   parameterEncoding    Name of parameter encoding to be used with    -->
<!--                        CGI servlet.                                  -->
<!--                        [System.getProperty("file.encoding","UTF-8")] -->
<!--                                                                      -->
<!--   passShellEnvironment Should the shell environment variables (if    -->
<!--                        any) be passed to the CGI script? [false]     -->
<!--                                                                      -->
 <servlet>
      <servlet-name>cgi</servlet-name>
      <servlet-class>org.apache.catalina.servlets.CGIServlet</servlet-class>
      <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
      </init-param>
      <init-param>
        <param-name>cgiPathPrefix</param-name>
        <param-value>WEB-INF/cgi</param-value>
      </init-param>
       <load-on-startup>5</load-on-startup>
 </servlet>
    ...
  <!-- The mapping for the CGI Gateway servlet -->
  <servlet-mapping>
      <servlet-name>cgi</servlet-name>
      <url-pattern>/cgi-bin/*</url-pattern>
  </servlet-mapping>
```

2. In the `$CATALINA_HOME/conf` directory, modify the default global `context.xml` file to set privilege levels for all the running applications. See the bold modification in the following example:

```
<Context privileged="true">
    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>
    <!-- Uncomment this to disable session persistence across Tomcat restarts -->
    <!--
    <Manager pathname="" />
    -->
</Context>
```

3. Restart Tomcat to cause the changes to be reread. Now Tomcat should serve up CGI scripts from the directory (typically, `/WEB-INF/cgi`) defined in the `cgiPathPrefix` element in `web.xml`.

To enable CGI support for selected Web applications, you need to:

1. Add the `<servlet>` and `<servlet-mapping>` elements (see Step 2 in the preceding list) for the CGI servlet shown above to the `WEB-INF\web.xml` file of the specific application.
2. Add `privileged="true"` to the `<context>` element in the `META-INF/context.xml` of the specific application.

If your system requires extensive CGI support, you should seriously consider front-ending Tomcat with a Web server (such as Apache Web Server or Microsoft's IIS). The CGI implementation on these servers is highly optimized and the implemented security measures are significantly more mature and robust. See the section "Front-Ending Tomcat 6 with a Web Server" later in this chapter.

Configuring Tomcat for SSI Support

Tomcat 6 support SSIs (server-side includes). SSI enables the adding of directives to HTML pages that are evaluated when the pages are served to the browser, and they are a popular mechanism for adding dynamic content. You have a choice of two mechanisms to support SSI in Tomcat 6:

- Via a servlet
- Via a filter

The servlet (`org.apache.catalina.ssi.SSIServlet`) simulates the way in which a Web server would handle an SSI in Web pages. The filter (`org.apache.catalina.ssi.SSIFilter`) basically does the same thing, but is a little more flexible and efficient in operation. For example, with the SSI filter, it is possible to conditionally process SSI based on the MIME type of a resource; it is also possible to include SSI within scripts processed by other components, such as a JSP page.

However, SSI is disabled in the default Tomcat configuration for security reasons. For instance, SSI could be used to execute external programs, and thus bypass the security policies defined in the `catalina.policy` file. More on these security policies is covered in Chapter 14.

Configuring the Tomcat 6 SSI Servlet

Enabling server-wide SSI support, via the Tomcat 6 SSI servlet, requires the following steps.

There are performance and security issues with Server Side Includes (SSI), and unless you have to support legacy application code, you should not enable it in a production environment. See Chapter 14 for more details.

1. Uncommment the `servlet` and `servlet-mapping` settings for SSI in `CATALINA_HOME/conf/web.xml`. These setting are commented by default. The sample settings are shown after the following descriptions of some configurable parameters:
   - `buffered`: Enables (1) or disables (0) buffered output from the SSIServlet
   - `debug`: The debug level to be enabled
   - `expires`: The expiry time, in seconds, for a Web page with SSIs
   - `isVirtualWebappRelative`: Virtual paths to be relative to context root (1) or server root (0)
   - `inputEncoding`: Text encoding for the page being processed for SSI (default OS encoding)
   - `outputEncoding`: Text encoding used for output of SSI processing (UTF-8)

```
<!--                                                                    -->
<!-- Server Side Includes processing servlet, which processes SSI       -->
<!-- directives in HTML pages consistent with similar support in web    -->
<!-- servers like Apache.  Traditionally, this servlet is mapped to the -->
<!-- URL pattern "*.shtml".  This servlet supports the following        -->
<!-- initialization parameters (default values are in square brackets): -->
<!--                                                                    -->
<!--   buffered            Should output from this servlet be buffered? -->
<!--                       (0=false, 1=true)  [0]                       -->
<!--                                                                    -->
<!--   debug               Debugging detail level for messages logged   -->
<!--                       by this servlet.  [0]                        -->
<!--                                                                    -->
<!--   expires             The number of seconds before a page with SSI -->
<!--                       directives will expire.  [No default]        -->
<!--                                                                    -->
<!--   isVirtualWebappRelative                                          -->
<!--                       Should "virtual" paths be interpreted as     -->
<!--                       relative to the context root, instead of     -->
<!--                       the server root?  (0=false, 1=true) [0]      -->
<!--                                                                    -->
<!--   inputEncoding       The encoding to assume for SSI resources if  -->
<!--                       one is not available from the resource.      -->
<!--                       [Platform default]                           -->
<!--                                                                    -->
<!--   outputEncoding      The encoding to use for the page that results -->
<!--                       from the SSI processing. [UTF-8]             -->
<!--                                                                    -->
<!--                                                                    -->
<!-- IMPORTANT: To use the SSI servlet, you also need to rename the     -->
```

(continued)

```
<!--           $CATALINA_HOME/server/lib/servlets-ssi.renametojar file   -->
<!--           to $CATALINA_HOME/server/lib/servlets-ssi.jar             -->
  <servlet>
      <servlet-name>ssi</servlet-name>
      <servlet-class>
        org.apache.catalina.ssi.SSIServlet
      </servlet-class>
      <init-param>
        <param-name>buffered</param-name>
        <param-value>1</param-value>
      </init-param>
      <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
      </init-param>
      <init-param>
        <param-name>expires</param-name>
        <param-value>666</param-value>
      </init-param>
      <init-param>
        <param-name>isVirtualWebappRelative</param-name>
        <param-value>0</param-value>
      </init-param>
      <load-on-startup>4</load-on-startup>
  </servlet>
  ...
  <!-- The mapping for the SSI servlet -->
  <servlet-mapping>
      <servlet-name>ssi</servlet-name>
      <url-pattern>*.shtml</url-pattern>
  </servlet-mapping>
```

2. In the `$CATALINA_HOME/conf` directory, modify the default global `context.xml` file to set privilege levels for all the running applications. See the bold modification in the following example:

```
<Context privileged="true">
    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>
    <!-- Uncomment this to disable session persistence across Tomcat restarts -->
    <!--
    <Manager pathname="" />
    -->
</Context>
```

3. Restart Tomcat to cause the changes to be reread. Now Tomcat should handle SSIs in all Web pages that end with `*.stml`.

To enable Servlet-based SSI support for selected Web applications, you need to do the following:

1. Add the `<servlet>` and `<servlet-mapping>` elements (see Step 2 in the preceding list) for the SSI servlet shown previously to the `WEB-INF\web.xml` file of the specific application.

2. Add `privileged="true"` to the `<context>` element in the `META-INF/context.xml` of the specific application.

As is the case with CGI support, if your system requires extensive SSI support, you should seriously consider front-ending Tomcat with a Web server for better security and support. See the section "Front-Ending Tomcat 6 with a Web Server" later in this chapter.

Configuring the Tomcat 6 SSI Filter

Instead of using the Tomcat SSI servlet, you may want to use the SSI filter instead. The filter enables more selective SSI processing (by MIME type, on JSPs, and so on). Only one of the two should be enabled for SSI processing.

Enabling SSI support in Tomcat requires the following steps:

1. Uncomment the `filter`, `filter-mapping`, and `mime-mapping` settings for the SSI filter in `CATALINA_HOME/conf/web.xml`. These setting are commented by default. The sample settings are shown after the following descriptions of some configurable parameters:

   - `contentType`: Use this regular expression to determine the content type that must be matched before SSI processing is performed on the request. You can configure this to match JSP, JavaScript file, and so on: `text/x-server-parsed-html(;.*)?`.
   - `debug`: The debug level to be enabled.
   - `expires`: The expiry time, in seconds, for a Web page with SSIs.
   - `isVirtualWebappRelative`: Virtual paths to be relative to context root (1) or server root (0).

```
<!-- Server Side Includes processing filter, which processes SSI          -->
<!-- directives in HTML pages consistent with similar support in web      -->
<!-- servers like Apache.  Traditionally, this filter is mapped to the    -->
<!-- URL pattern "*.shtml", though it can be mapped to "*" as it will     -->
<!-- selectively enable/disable SSI processing based on mime types. For   -->
<!-- this to work you will need to uncomment the .shtml mime type         -->
<!-- definition towards the bottom of this file.                          -->
<!-- The contentType init param allows you to apply SSI processing to JSP -->
<!-- pages, javascript, or any other content you wish.  This filter       -->
<!-- supports the following initialization parameters (default values are -->
<!-- in square brackets):                                                 -->
<!--                                                                      -->
<!--   contentType         A regex pattern that must be matched before    -->
<!--                       SSI processing is applied.                     -->
<!--                       [text/x-server-parsed-html(;.*)?]              -->
<!--                                                                      -->
<!--   debug               Debugging detail level for messages logged     -->
<!--                       by this servlet.  [0]                          -->
<!--                                                                      -->
<!--   expires             The number of seconds before a page with SSI   -->
<!--                       directives will expire.  [No default]          -->
<!--                                                                      -->
```

(continued)

```
<!--   isVirtualWebappRelative                                          -->
<!--                       Should "virtual" paths be interpreted as     -->
<!--                       relative to the context root, instead of     -->
<!--                       the server root?  (0=false, 1=true) [0]      -->
<!--                                                                    -->
<!--                                                                    -->
  <filter>
      <filter-name>ssi</filter-name>
      <filter-class>
        org.apache.catalina.ssi.SSIFilter
      </filter-class>
      <init-param>
        <param-name>contentType</param-name>
        <param-value>text/x-server-parsed-html(;.*)?</param-value>
      </init-param>
      <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
      </init-param>
      <init-param>
        <param-name>expires</param-name>
        <param-value>666</param-value>
      </init-param>
      <init-param>
        <param-name>isVirtualWebappRelative</param-name>
        <param-value>0</param-value>
      </init-param>
  </filter>
   ...
  <filter-mapping>
     <filter-name>ssi</filter-name>
     <url-pattern>*.shtml</url-pattern>
  </filter-mapping>
  ...
  <mime-mapping>
      <extension>shtml</extension>
      <mime-type>text/x-server-parsed-html</mime-type>
  </mime-mapping>
```

2. Restart Tomcat to cause the changes to be reread. Now Tomcat should handle SSIs in all Web pages that end with `*.stml`, as well as any content that matches the regular expression that you have configured with the `contentType` initialization parameter.

Running Tomcat Behind a Proxy Server

A common deployment scenario involves running Tomcat behind a proxy server. In this kind of environment, the host name and port number that should be returned to the client in the HTTP response should be those specified in the request, and not the actual host name and port on which Tomcat is running. These are controlled via the `proxyName` and `proxyPort` attributes discussed earlier. These attributes affect the values returned for the `request.getServerName()` and `request.getServerPort()` Servlet API calls.

Apache's HTTP server can be used as the proxy server. If so, Apache's proxy module (`mod_proxy`) is configured to pass on the servlet requests to the Tomcat server:

```
# Load mod_proxy
LoadModule proxy_module libexec/mod_proxy.so
# AddModule only required for Apache 1.x, not 2.x
AddModule mod_proxy.c
# Pass all requests for the context path '/servlets' to Tomcat running
# at port 8080 on host 'hostname'
ProxyPass /servlets http://hostname:8080/servlets
ProxyPassReverse /servlets http://hostname:8080/servlets
```

On the Tomcat side, following is the configuration in `server.xml` for the standard HTTP/1.1 Connector:

```
<Connector
               port="8080"
               proxyName=www.mydomain.com
               proxyPort="80"
               ...
/>
```

If the `proxyName` and `proxyPort` attributes were not specified, the response message would have indicated that it came from *host name* (i.e., the host on which Tomcat is installed), and 8080 instead of `www.mydomain.com` and port 80.

Typically, Apache logs incoming requests, so logging shouldn't be enabled on Tomcat — to avoid duplicate access logging.

Performance Tuning

The default HTTP/1.1 Connector for Tomcat 6 is already a mature, high-performance Connector suitable for service of both static resources and dynamic output from JSP and servlets. For new Ajax-styled applications requiring long-lasting sessions between client and server, the non-blocking NIO Connector can provide a workable, high-performance solution. In hardcore production environments where applications are unlikely to change frequently, the use of the native APR Connector provides a high performance, scalable, and potentially accelerated solution.

Tunable Configuration Attributes

Each and every deployment configuration is different, and the performance tuning required may also differ greatly. This section attempts to provide some words of wisdom from our own attempts to tune and optimize performance of Tomcat 6 servers. The configuration sections previously covered in this chapter discussed some of the attributes that affect performance characteristics of the HTTP Connectors. The following are some tuning tips for Tomcat 6:

- Set `tcpNoDelay` to `true`: When this attribute is set to `true`, it enables the `TCP_NO_DELAY` network socket option. This improves performance as it disables the Nagle algorithm, which is used to concatenate small buffer messages, which decreases the number of packets sent over the network. While this may result in better response time in a non-interactive network application

because it enables greater throughput, it results in slower response times in interactive client-server environments (such as a Web browser interacting with the Web server).

- `maxKeepAliveRequest`: This attribute controls the "keep-alive" behavior of HTTP requests, enabling persistent connections (that is, multiple requests to be sent over the same HTTP connection). It specifies the maximum number of requests that can be pipelined until the connection is closed by the server. The default value of `maxKeepAliveRequest` is `100`, and setting it to 1 disables HTTP keep-alive behavior and pipelining.
- Tune the `socketBuffer` parameter: As mentioned, this specifies the size, in bytes, of the buffer to be used for socket output buffering.
- Set `enableLookups` to `false`: Setting `enableLookup` to `false` disables DNS lookups for the `request.getRemoteHost()` API calls. This improves performance by decreasing the time required for the lookup.
- Use a thread pool: Tomcat is a multi-threaded Servlet container, and each incoming request requires a Tomcat thread to handle it. Using a thread pool is thus very important for performance. In Tomcat 6, three thread pool–related attributes can be tuned. Setting these to appropriate values varies according to the Web site load and the server machine's characteristics:
  - `maxThreads`: This is the maximum number of threads allowed. This defines the upper boundary to the concurrency, as Tomcat will not create any more threads than this. If there are more than `maxThreads` requests, they will be queued until the number of threads decreases. Increasing `maxThreads` increases the capability of Tomcat to handle more connections concurrently. However, threads use up system resources. Thus, setting a very high value might degrade performance, and could even cause Tomcat to crash.
  - `maxSpareThreads`: This is the maximum number of idle threads allowed. Any excess idle threads are shut down by Tomcat. Setting this to a large value is not good for performance; the default (`50`) usually works for most Web sites with an average load. The value of `maxSpareThreads` should be greater than `minSpareThreads`, but less than `maxThreads`.
- `minSpareThreads`: This is the minimum number of idle threads allowed. On Tomcat startup, this is also the number of threads created when the Connector is initailized. If the number of idle threads falls below `minSpareThreads`, Tomcat creates new threads. Setting this to a large value is not good for performance, as each thread uses up resources. The default (`4`) usually works for most Web sites with an average load. Typically, sites with "bursty" traffic would need higher values for `minSpareThreads`.
- JVM memory settings: Another limiting factor here is the JVM memory settings. To add a larger number of threads, Tomcat startup scripts (`catalina.bat/catalina.sh`) must be modified to pass JVM-specific parameters (such as `-Xms` and `-Xmx` to set the initial and maximum heap size). Refer to your JVM documentation for additional information.

TCP/IP Stack Tuning Tips

The following tuning tips are for the TCP/IP stacks on the operating system, and can frequently benefit a Tomcat server running over it.

Windows XP or Server 2003 TCP Stack Tuning

Tuning the `TcpTimedWaitDelay` determines the time the TCP implementation will wait before recycling a closed connection. By keeping connections in the `TIME_WAIT` state, a connection to the same endpoint can

be established faster than creating a brand new connection. Yet, having this parameter set too long may result in massive resource consumption when a server is rapidly hit by many short-duration connections. The default on Windows is set to 4 minutes. You should seek a balance for your own server and request a load profile. Adjusting this parameter requires the use of a registry editor; locate and edit the parameter under the key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TCPIP\Parameters`.

Another TCP parameter that you may want to tune is `TcpMaxDataRetranmission`. This parameter governs the number of retransmission attempts for a segment that is not acknowledged. This ability to retransmit is great for unreliable links, but you may want to tune this value down if you are on a solid network. The default value is set to 5.

Linux TCP Stack Tuning

The equivalent parameter to `TcpTimeWaitDelay` in Linux is located under `/proc/sys/net/ipv4/tcp_fin_timeout`. The default value is set at 4 minutes.

A similar parameter to `TcpMaxDataRetranmission` in Linux is located under `/proc/sys/net/ipv4/tcp_keepalive_probes`. This value governs the number of keep-alive probes sent on an unacknowledged connection before abandoning the connection. The default value is set at `9`.

Front-Ending Tomcat 6 with a Web Server

Even experts disagree on the best overall choice of configuration for Tomcat 6. Perhaps this is an indication that one size does not fit all.

You can select one of the Connectors examined in this chapter and run Tomcat 6 in standalone mode, or you can use a production-grade Web server (such as Apache or Microsoft's IIS) as a front-end to Tomcat, redirecting only JSP and servlet requests to Tomcat. The next two chapters cover these Web server front-end configurations.

Before the availability of the APR-based native code Connector, the stabilization of the NIO Connector, and the greatly improved performance of modern Java VMs, the choice was somewhat easier to make purely based on a performance argument. But with the availability of these optimized alternatives, one can no longer reach a conclusion based on this single argument. In fact, with these alternatives, Tomcat on a standalone configuration — without a Web server front-end — can be a highly performant and viable production configuration.

Your choice for the best configuration, then, is going to be dependent on the composition and the environmental needs of your particular application (or application mix if you are virtual hosting). The only reliable way to determine what is best for any particular application and/or application mix is to benchmark the performance against the different configurations (over a significant period of time and over a typical mix of loading profiles) and then base your decision on the result of these benchmarks.

Other than benchmarking your applications on the alternative configurations, the following are some general guidelines that may help with your decision:

- If your solution involves the configuration of a cluster of Tomcats, you should go with a Web server front-end, allowing it to redirect incoming requests to any machine in the cluster; this is covered in Chapter 17.

- If your production installation is particularly security sensitive, either Apache Web Server or IIS offers significantly better security support and more timely security updates. A large security community monitors 24/7 for newly discovered vulnerabilities for these platforms, and engineers the required patches to fix them.
- If your application or application mix is very light on dynamic content (JSP and Servlet) and heavy on static contents (images and static HTML pages), chances are good that a Web server front-end configuration will better serve your needs. However, even this can be disputed on lightly loaded Web sites because the standalone configuration is highly adaptive in these situations.
- If your overall Web strategy involves dynamic page generation technology beyond servlets and JSPs — for example, if part of your production deployment involves Perl, PHP, Python, and/or ASP — using a Web server front-end may be your only choice.

Summary

To conclude this chapter on Connectors, let's review some of the key points that have been discussed:

- HTTP connectors are Java classes that implement the HTTP protocol. For Tomcat, this class is invoked when there is an HTTP request on the Connector port.
- The default HTTP/1.1 Connector in Tomcat 6 is a mature Java-based implementation that has user testing since the 3.*x* version of Tomcat.
- Another Java-based HTTP Connector implementation, taking advantage of Java 5's NIO libraries, is also available as an alternative.
- A native code–based HTTP Connector, written using APR, is available as a performance/scalability-enhanced alternative. This connector features the integration of a scalable keep-alive connection poller, the use of the kernel mode `sendfile()` API to send large static files, and the integration of the well-known `OpenSSL` native library for handling SSL connections.
- Configuration for the default Java-based HTTP/1.1 Connector, the NIO Connector, and the APR Connector, including support for SSL and performance tuning, has been covered in detail in the chapter.
- The rich selection of HTTP Connectors covers a variety of differing hardware and operation configurations with high performance and scalable solutions. In former versions of Tomcat where the availability of different HTTP Connectors is limited, administrators often prefer a Web server front-end as the best choice for configuration. While this continues to be true in highly security-sensitive installations, most other installations are well served by the Connectors built into Tomcat 6.

Chapters 11 and 12 discuss the Web server Connectors that enable Tomcat 6 to be front-ended by Web servers such as Apache and IIS.

11

Tomcat and Apache HTTP Server

A typical scenario in production environments is to use Tomcat along with a Web server. In this scenario, the Web server is used as a frontend to Tomcat. The Web server serves all static content and Tomcat serves all dynamic content. Tomcat does have its own built-in HTTP server, but some administrators managing large Web server farms may insist on using a well-known and robust Web server to accept requests from the Internet. A number of Web servers can be used for this purpose (including Apache, IIS, and Netscape). This chapter describes the process of connecting the Apache Web server to Tomcat.

Tomcat can be integrated with Apache using the JK Connector. This chapter explains how to install and configure this Connector. The JK Connector uses the Apache JServ Protocol (AJP) for communication between Tomcat and the Apache Web server.

The following topics are covered in this chapter:

- The AJP (Apache JServ Protocol) Connector and the JK module (`mod_jk` module)
- The AJP Connector and the `mod_proxy` module
- Configuring Tomcat 6 with Apache 1.3.*x* using `mod_jk`
- Configuring Tomcat 6 with Apache 2.0.*x* using `mod_jk`
- Configuring Tomcat 6 with Apache 2.2.*x* using `mod_proxy`
- Configuration of SSL for Apache 2.2.*x* servers
- Load balancing multiple Tomcat instances with Apache

Each of the 1.3.*x*, 2.0.*x*, and 2.2.*x* series of the Apache server has slightly different architecture and feature sets. A later version does not necessarily displace another, and each version has its own strong following of server administrators. This chapter covers configurations of Tomcat 6 with

each series of Apache server. The techniques demonstrated are tested against Tomcat 6; Apache 1.3.37, 2.0.59, and 2.2.3; and `mod-jk2-2.0.43` with Sun's JDK-1.5_11. It is assumed your required version of the Apache Web server is installed, configured, and running on the system. Apache binaries are available for download at `apache.org/dist/httpd/`.

The complete Apache documentation is available at `http://httpd.apache.org/docs/`.

The AJP Connector Architecture

The integration between the Apache Web server and Tomcat is accomplished via specialized software modules that are added to both sides. These software modules communicate with one another over the network using a common protocol. This network protocol is called AJP, covered later on in this chapter.

The software modules that are involved on the side of Apache Web server are either `mod_jk` or `mod_proxy`. Both are native code extension modules written in C/C++; on the side of the Tomcat server, the software module is a specialized AJP Connector written in Java.

The Native Code Apache Modules

Figure 11-1 shows how the native code Apache module, either `mod_jk` or `mod_proxy`, works with Tomcat 6. All requests first come to the Apache Web server. The Apache Web server accepts and processes any static resource requests, such as requests for static HTML pages or graphical images. With the help of either `mod_jk` or `mod_proxy`, the Apache Web server redirects requests for any JSP or servlet component to Tomcat server instance(s). This redirection is performed by sending the request over the network (usually a LAN connecting the Apache and Tomcat servers together) using the AJP protocol.

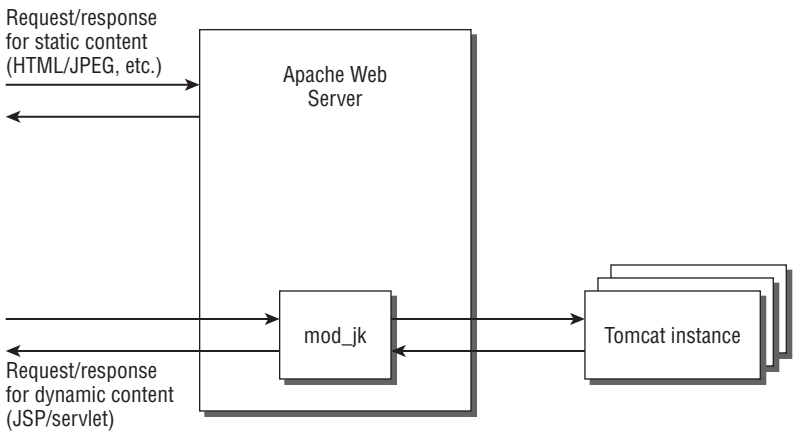

Figure 11-1: Connecting the Apache Web server to Tomcat 6

There may be one or more network-connected instances of Tomcat for serving these client requests. Each of these Tomcat instances has an AJP Connector configured. The AJP Connector monitors the network (LAN) for requests redirected by the Apache Web server. This AJP Connector receives and

processes the requests and generates the appropriate response. Finally, Tomcat 6 sends this response back to the Apache Web server, through the AJP Connector, to the waiting native code module on the Apache Web server side. Once the response is received, the Apache Web server in turn forwards the response to the client.

The native code module on the Apache Web server talks to the AJP Connector on Tomcat 6 via the AJP protocol. The AJP protocol is an optimized request/response transfer protocol between a source of request (an Apache Web server) and one or more processors of those requests (a cluster of Tomcat 6 servers).

> *The native code Apache Web server module (*`mod_jk` *or* `mod_proxy`*) usually comes as a DLL on a Windows platform, as a shared object module (.so file for Unix or Linux), or statically linked into the Web server executable binary. The AJP Connector on the Tomcat 6 end is a 100 percent Java implementation, and comes standard as part of the Tomcat 6 distribution.*

The Apache JServ Protocol

AJP is a packet-oriented, TCP/IP-based protocol. It provides a communication channel between Apache and the running instances of Tomcat. There are various versions of the AJP protocol (AJP1.0, AJP1.1, AJP1.2, and AJP1.3), as described briefly in Chapter 4. Tomcat 6 supports AJP1.3, which is the well-tested version. Some of its major features include the following:

- Good performance on fast networks (such as gigabit ethernet); AJP 1.3 does a good job of compressing elements of the protocol that are transmitted over the wire, thus reducing the protocol overhead.
- Support for SSL, encryption, and client certificates.
- Support of clustering by forwarding requests to multiple Tomcat 6 servers.

One of the ways in which AJP reduces connection latency (a factor that adversely affects performance) is by making the Web server reuse already open TCP-level connections with Tomcat. This saves the overhead of opening new socket connections for each request. This concept is similar to that of a connection pool. In the request-response cycle, when a connection is assigned to a particular request, it will not be reused until that request-response cycle is completed.

The AJP Connector

Tomcat 6 provides an AJP Connector implementation for the AJP 1.3 protocol as a Java class included with the Tomcat distribution.

The `$CATALINA_HOME/conf/server.xml` configuration file contains the following entry for the Connector:

```
<!-- Define an AJP 1.3 Connector on port 8009 -->
    <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

The `protocol="AJP/1.3"` attribute indicates to Tomcat that you are configuring the AJP Connector. You use this connector on the Tomcat server instances to listen for incoming AJP requests over the LAN, sent by the Apache Web server.

Apache Web Server Frontend or Tomcat Standalone

For a long period of time in the recent past, the standalone server configuration of Tomcat was criticized by some users as lacking performance. One of the key objectives during the development of Tomcat 5, leading to Tomcat 6, was to improve the performance of the all-Java server implementation.

Judging from user feedback, it appears that Tomcat 6 (and the later revisions of Tomcat 5) has achieved this objective. In fact, if you are running on one server only with moderate load (no clustering), you are likely to be able to get the same or better performance if you use just the Tomcat 6 server's own HTTP/1.1 Connectors (described in Chapter 10) instead of running an instance of the Apache Web server connected to Tomcat using the AJP protocol.

There are, however, still a lot of reasons beyond performance that Apache Web servers continue to be the preferred frontend for Tomcat 6. The following are just a sampling of the mix:

- **Security:** Security is a serious concern in this day and age; the thousands of security vulnerabilities of a complex piece of software, such as a Web server, need to be discovered, fixed, and retested constantly. With the Apache Web server, this is happening daily and continuously, with a very large base of security experts working around the clock. Because a limited number of Tomcat 6 servers are directly connected to the Internet, the community hasn't invested as much research and effort in security for Tomcat 6. Some may argue that this actually makes the Tomcat 6 server less of an attack target and that it is therefore a safer alternative.
- **Scaling out via clustering:** If you need to run a cluster of Tomcat servers to take on the computing demands of your application, your only choice may be the AJP Connector. In Chapter 17, we talk extensively about clustering and how to use a frontend Apache Web server to forward application requests to a cluster of Tomcat 6 servers. This configuration spreads the computing load across multiple physical machines, and enables the cluster to handle a larger volume of simultaneous requests than one standalone machine can.

Understanding Tomcat Workers

Before we configure the modules and AJP Connector, you must understand the concept of the *worker*. A worker represents a running instance of Tomcat. A worker serves the requests for all the dynamic Web components. The requests may come from the Apache Web server or directly from a client. In many cases, there is only a single Tomcat process or instance. However, sometimes multiple workers must be running in a Tomcat 6 cluster to implement load balancing or site partitioning (mainly required for sites with heavy traffic). This topic is discussed in the section "Tomcat Load Balancing with Apache," later in this chapter.

Each worker is identified by a unique host name or a unique combination of IP address and port number. The host refers to the machine name on which the given Tomcat instance is running, and the port refers to the (AJP protocol) port on which that Tomcat instance is listening for any incoming requests.

Multiple Tomcat Workers

There are a number of situations in which you may need to use multiple workers, including the following:

- **When you want different Web application contexts to be served by different Tomcat workers:** In production, this may happen when some workers are privy to specialized hardware connectivity or setup (say, access to a Fiber Channel Storage Area Network). During testing, this setup can provide a development environment in which all the developers share the same Web server but have a dedicated Tomcat worker.
- **When you want different virtual hosts served by different Tomcat processes to provide a clear separation between sites belonging to different entities:** This is useful when hardware is paid for by different clients in a shared hosting environment, or when stability of operation requires that applications from different sources be isolated physically.
- **When you want to service more requests than the capacity of a single physical machine allows:** You can implement a cluster of machines with Tomcat workers running on them and distribute the load between them.

Configuring Apache Server to Work with Multiple Tomcat Workers — the workers.properties File

In order for the Apache Web server to forward incoming requests to the Tomcat 6 instances, it must know where they are located on the network. To describe where the Tomcat servers are located on the network, and to provide specific instructions on how to work with each Tomcat instance, you need to create a configuration information file called `workers.properties`. This file is placed in the `conf` directory of your Apache Web server. `mod_jk` will scan the `conf` directory for the `workers.properties` file during startup, and then process the configuration file and instantiate connections to the workers. The following sections explore this `workers.properties` file in more detail.

Format of the workers.properties File

The general flow of a `workers.properties` file is:

1. Description of the list of workers
2. Description of each of the worker instance in the list from the previous step

To describe the list of workers operating with an Apache server, you can use one or more of the following statements.

```
worker.list =  <a comma separated list of worker names>
```

For example, the following line in `workers.properties` describes two workers named `testworker1` and `testworker2`:

```
worker.list = testworker1,   testworker2
```

If you have many workers, you don't have to specify them all in one line; you can use multiple `worker.list` definitions. Using multiple definitions is also handy when there are certain logical groupings of workers.

To describe each instance of the workers named in the `worker.list` statement(s), you can use as many statements of the following format as you need:

```
worker.<worker name>.<property> = <property value>
```

The `<worker name>` is significant and is usually the same name configured as `jvmRoute` on the Tomcat-side configuration.

For example, the following line in `workers.properties` assigns the IP address 192.168.1.128 to the host property of the `testworker1` worker instance:

```
worker.testworker1.host = 192.168.1.128
```

Many different properties can be set on each worker instance. The following sections describe these properties.

Types of Workers

One of the more important properties for a worker instance that can be set in the `worker.properties` file is the type of the worker. Each defined Tomcat worker must be assigned a type. Tomcat 6 integration with the `mod_jk` Connector supports the following types of workers:

- `ajp13`: This type of worker represents a running Tomcat instance. There are various possible attributes for this worker. The main attributes include `tomcatId` (which represents the identity of the Tomcat instance), `channel` (which indicates the communication channel associated with this worker), and `max_connections` (which is used to specify the maximum number of connections). By default, the maximum number of connections is unlimited. The default port for AJP 1.3 is 8009.
- `lb`: This type of worker is used for *load balancing*. In a load-balancing scenario, the worker doesn't actually process any requests. Rather, it handles the communication between a Web server and other defined Tomcat workers of type `ajp13`. The worker supports round-robin load balancing with a certain level of fault tolerance. One of the main properties for this worker is `worker`, which indicates the name of the worker to be used as a load balancer. A number of attributes are provided by the `lb` worker. Some of the attributes are explained later in this chapter.
- `status`: This is a special type of worker that is used to show useful information about how the load among the various Tomcat workers is distributed. It does not process any request, and is not associated with any Tomcat worker instances. To use it, add a mapping of URL using the `JkMount` directive assigned to this worker. Its use is explained in the section "Tomcat Load Balancing with Apache," later in this chapter. The `jkstatus` Web page (again, described later) typically displays some very vital information, including the available number and names of workers, the associated `lb_factor`, and their locations. It also indicates the number of requests served by a specific worker and any context mappings served. If any worker goes into an error state, this can be easily detected from this page.
- `jni`: Used in-process, this worker handles the forwarding of requests to in-process Tomcat workers using JNI. In the in-process mode, the Tomcat Web container and the Web server share the same memory address space. They communicate via interprocess communication. This worker holds the details of the Tomcat class to start up, and which parameters to pass. There are two predefined `jni` workers: `onStartup` and `onShutdown`. These are executed during the startup and shutdown phase of the connector, respectively. Both must exist in the configuration in order to be able to start and shut down Tomcat. See the accompanying box about in-process workers.
- `ajp12`: A worker supporting the AJP 1.2 protocol. This is a legacy protocol, supported by `mod_jk` only for backwards compatibility purposes. This protocol should not be used with Tomcat 6 integration.

For example, the following line in the `worker.properties` file sets the type of `testworker1` to `ajp13`, asking `mod_jk` to forward a request to the `testworker1` Tomcat 6 instance using the AJP 1.3 protocol:

```
worker.testworker1.type = ajp13
```

In addition to the `type` property described here, other frequently used worker properties are described in the next section.

> **An *in-process worker* is a Tomcat instance running inside a Java VM that is running within the same process, and shares the same memory space, as the Apache Web server. This is a specialized configuration that is seldom used nowadays. The disadvantages of this configuration include the fact that any instability in either the Tomcat or the Apache Web server can affect the other, and that the computing facilities of the same physical machine must be shared between the Web server and the Tomcat/Java VM, making it completely non-scalable. In-process worker configuration is also difficult to configure, troubleshoot, and maintain. This book will not cover in-process workers.**

Tomcat-based configuration of an individual worker instance is described later, in the section "Configuring Tomcat Workers."

Other Worker Properties

Depending on the type of a worker instance, additional properties can be defined for the instance in the `worker.properties` file. If the worker instance is of `ajp13` type, the following additional properties are available:

- `host`: The host where the worker Tomcat 6 instance resides. You can specify a resolvable host name or an IP address here.
- `port`: The port the AJP 1.3 Connector of the Tomcat worker instance is listening to for incoming requests. The default is 8009 for AJP 1.3 connections.
- `connection_pool_size`: The number of connections used for this worker to be kept in a connection pool. The connection pool enhances response time because re-creation of a socket is an expensive operation. If there are already connections to a worker in the connection pool, the Apache Web server can forward requests to the worker immediately, without having to create a new connection. The default value depends on the server. On Apache 1.3 and 2.*x* prefork, this should be set to 1 because they are not multithreaded. On 2.*x* mpm (**m**ulti-**p**rocessing **m**odules — a new 2.*x* feature designed to take maximum advantage of the underlying operating system), multithreaded implementation, this is set by default to the threads-per-child value.
- `connection_pool_minsize`: The minimum number of connections kept in the connection pool. By default, it is set to `pool_size/2` for multithreaded Web servers.
- `connection_pool_timeout`: The number of seconds that connections to this worker should be left in the connection pool before expiry.
- `mount`: The context paths that are serviced by the worker. The paths should be separated by blank space. This property can be used instead of, or in addition to, the `JkMount` directive in the `httpd.conf` file (described later).

- `retries`: Controls the number of times `mod_jk` will retry a worker when an error is returned during request forwarding. A small delay is inserted between each retry (100ms). The default is `3`.
- `socket_timeout`: The default is `0`, which means to wait indefinitely. This timeout controls how long, in seconds, `mod_jk` will wait for a worker to respond on a socket before indicating an error.
- `socket_keepalive`: Indicates if the connection to the worker should be subjected to keep alive: `1` for `true`, and `0` for no keep alive on the connection. The default is `0`. The `keepalive` attribute is useful when a firewall sitting between the Web server and the worker may drop connections that have been inactive for a long time.
- `lbfactor`: An integer indicating the load-balance factor used by a load balancer to distribute work between multiple instances of Tomcat 6 servers. A more detailed description of load balancing, and how this factor is used, is available in the section "Tomcat Load Balancing with Apache."

For example, the following `worker.properties` file segment configures a Tomcat 6 instance called `testworker1` at 192.168.1.128, listening for AJP 1.3 requests on port 9009. The load-balancing factor is set at 20 for this worker. Outstanding connections to this worker instance are limited to 5, and each connection is maintained in the pool for up to 5 minutes (300 seconds).

```
worker.list = testworker1
worker.testworker1.type = ajp13
worker.testworker1.host = 192.168.1.128
worker.testworker1.port = 9009
worker.testworker1.connection_pool_size = 5
worker.testworker1.connection_pool_timeout = 300
```

The properties available for configuring an `lb worker` are listed next. Note that an `lb worker` is not a physical Tomcat 6 instance. In fact, it does not have a host or port property and does not cause any network traffic. Instead, it is the control center for configuring the load-balancing behavior for a group of physical workers.

- `balance_workers`: A list of workers to balance the load between, when using this load balancer. Any worker that appears here must not appear in a `worker.list`; instead, the `worker.list` should contain the name of the load-balancer worker.
- `lock`: The default is `O` (Optimistic lock). The property controls how the load balancer accesses shared runtime memory containing stats to determine which worker to use. In `O` mode, there is no lock on the shared runtime memory used, and the stats value can change during the determination. It can also be set to `P` (Pessimistic lock) — in this case, the selection is more accurate but performance during load balancing may suffer slightly.
- `method`: Can be set to `R`, `T`, or `B`. The default is `R`. This controls the selection of worker for request forwarding. When set to `R` (Requests), the worker to use is based on the number of requests forwarded. When set to `T` (Traffic), the worker to use is based on the traffic that had been sent to the workers. When set to `B` (Busy-ness), the worker to use is based on the load by dividing the number of concurrent requests by the load factor.
- `secret`: Sets a default secret password for all workers.
- `sticky_session`: The default is set to `1`. This tells the `mod_jk` to respect the session ID in the request, and ensures that the same session is always serviced by the same worker instance. If you set it to `0`, the request from the same session may be forwarded to any of the worker instances. You should set it to `0` only if the session manager deployed has some way of providing shared session information across workers.

- `sticky_session_force`: The default is set to `0`. This is used for fail-over. If the value is `0` and the `SESSION_ID` is in an error state, a failover to another worker will occur. If this is set to `1`, a standard server error will be returned to the client.

Connecting Tomcat with Apache

Connecting Tomcat with Apache Web server first requires you to select a native code module to use with the Apache Web server. The exact native code module to use to integrate Tomcat 6 with Apache Web server will depend on the release version of the Apache Web server that you are using.

The following table provides a compatibility matrix between the current versions of Apache Web servers and the available native code modules for Tomcat 6 integration via the AJP 1.3 protocol.

Apache Web Server Releases/ Native Code Modules	**1.3.*x***	**2.0.*x***	**2.2.*x***
`mod_jk`	Yes	Yes	Yes
`mod_proxy`	No	Yes (need recompile from code out of 2.2.*x*)	Yes

Chances are good that your choice of an Apache Web server version is dictated by existing system requirements, or perhaps availability of administration expertise. However, if you do have the luxury of selecting a Web server from scratch and no legacy system/application compatibility requirements — the selection of version 2.2.*x* with `mod_proxy` support can provide you with the most feature-rich and secure combination available to date.

From the previous table, the ideal native module to use is `mod_jk` for Apache Web server, version 1.3.*x* and 2.0.*x*; while `mod_proxy` is the best matched choice for Apache Web server, version 2.2.*x*.

After you have selected a native code module, connecting Tomcat with Apache involves a series of steps, broken down to two distinct flows of configurations:

- Configuration on the Tomcat 6 side
- Configuration on the Apache Web server side

The following two sections look at the configuration steps required.

Tomcat 6 Configuration

On the Tomcat instance(s), enable the Tomcat 6 AJP 1.3 Connector in `server.xml`.

During this process, the file that needs to be modified is `server.xml`. This is Tomcat's main configuration file, located in `<CATALINA_HOME>/conf`. This is the file in which the AJP Connector is configured.

The next section shows you how to configure the AJP Connector.

Configuring the AJP 1.3 Connector in server.xml

The default `server.xml` file already has a configuration entry for the AJP 1.3 Connector, but it may be commented out. Uncomment this tag if commented. This is what you should see:

```
<!-- Define an AJP 1.3 Connector on port 8009 -->
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

These lines represent an AJP Connector that will use AJP version 1.3 and listen on AJP's default port (8009). The `port` attribute is configurable, and is the port on which Tomcat listens for AJP requests. Most of the attributes for the default Java HTTP/1.1 Connector mentioned in Chapter 10 also apply to the AJP Connector. Here are some of the key configurable attributes for this Connector:

- `enableLookups`: If set to `true`, calls to `request.getRemoteHost()` will perform DNS lookup to return the actual host name of the remote client. Setting this to `false` will skip the DNS lookup and return the IP address as a string (thereby improving performance). By default, DNS lookups are disabled.
- `redirectPort`: If a request is received for which a matching `<security-constraint>` requires SSL transport, Tomcat automatically redirects the request to the port number specified here.
- `scheme`: Set this attribute to the name of the protocol you want to have returned by calls to `request.getScheme()`. For example, you would set this attribute to `https` for an SSL Connector. The default value is `http`.
- `secure`: If set to `true`, calls to `request.isSecure()` will return `true` where the Connector is SSL-enabled. The default value is `false`.
- `protocol`: This attribute value must be `AJP/1.3` in order to select the AJP Connector.

Apache Web Server Configuration

On the Apache Web server side, you may have a choice of using either `mod_jk` or `mod_proxy`, depending on the version of Apache server you are using.

In either case, you need to perform the following steps:

1. Install the Apache `mod_jk` or `mod_proxy` binary module to your Apache Web server. In some cases, the binary is not available and you must build/compile the binaries yourself from a download source code.
2. Add directives in the Apache Web server configuration to load the `mod_jk` module (`httpd.conf`).
3. Configure the `worker.properties` file describing the Tomcat worker instances that are available.

The files that are created or modified during this configuration include:

- `httpd.conf`: This is Apache's main configuration file and is located under `<APACHE_HOME>/conf`. You add directives to load the `mod_jk` module to this file.
- `worker.properties`: This file specifies the communication channel details needed for connecting Tomcat with Apache. This file is located under `<APACHE_HOME>/conf`.

Some differences exist between the configuration of `mod_jk` versus `mod_proxy`. The next two sections cover the use of `mod_jk` first, and then the use of `mod_proxy`.

Using the mod_jk Module

You need to have the `mod_jk` module binaries before you can configure the Apache Web server for Tomcat connection.

Depending on the version of the Apache Web server that you are using, as well as the operating system platform (Windows or Linux), you may not be able to find precompiled binaries for `mod_jk` modules. The next section shows you where to look for `mod_jk` binaries, and how to build the binary from source if you need to.

Native Code mod_jk Binaries for Apache Server

The Tomcat Project manages a separate subproject for Tomcat Connectors. A rich repository of various combinations exists for `mod_jk` on many platforms. Both the source and binaries are available for download. You can get the latest binary version of the `mod_jk` Connector module from the following URL: `http://tomcat.apache.org/download-connectors.cgi`.

The binaries of `mod_jk` for Apache versions 1.3.*x* and 2.*x* are available at this URL.

First, locate the directory that corresponds to your operating system (Linux, Win32, Win64, and so on). Then select the directory of the version of `mod_jk` you want to download — at the time of writing it is jk-1.2.19. Next, check the list of downloads for `mod_jk-apache-x.x.xx.so`, making sure that the `x.x.xx` matches your version of the Apache Web server, and then copy the `.so` file to the `modules` directory of the Apache Web server. For example, if you are using Apache 1.3.37, make sure you find and download `mod_jk-apache-1.3.37.so`. If you are using Apache 2.0.58, make sure you download `mod_jk-apache-2.0.58.so`, and so on.

Copy the downloaded module file to the `<APACHE_HOME>/modules` directory.

> *Note that there exists a subdirectory for each operating system, and each version of the* `mod_jk` *module. Make sure you obtain the correct version of* `mod_jk`*, for the correct operating system, and supporting the correct version of Apache Web server for your own installation.*

If you cannot find a matching `mod_jk` binary for your version of the Apache Web server, you need to compile your own binary from source.

Building mod_jk on Windows

You will need to make sure you have Visual Studio .NET 200x installed; this is required (or at least a version of Visual C++ 6.x). Use the following steps to build `mod_jk` on the Windows platform:

1. Download the source of the Apache Web server, corresponding to the version of the server you are using, from `http://httpd.apache.org/download.cgi`.
2. Unpack the distribution into any convenient directory, such as `c:\apachesrc\server`.
3. Download the latest Connector source from `http://tomcat.apache.org/download-connectors.cgi`.

4. Unpack it to a convenient directory, such as `c:\apachesrc\mod_jk`.

5. Now, compile the Apache Web server source to generate the required linkage libraries. First change the working directory to the Apache Web server source code directory, and then open the Visual Studio .NET 200x command prompt (created as part of the Visual Studio installation). This command prompt sets all the required environment variables for Visual C compilation. From the command prompt, issue the command `nmake /f Makefile.win _apacher` from the Web server installation directory. This builds all the libraries that you will need to link `mod_jk` successfully for your server.

6. Set `%APACHE2_HOME%` (or `%APACHE1_HOME%` if you are using Apache 1.3.*x*) for the Apache source distribution (for example, `c:\apachesrc\server\httpd-2.2.3`).

7. Set `%JAVA_HOME%` for the JSE 5 installation (for example, `c:\jdks\jdk1510`).

8. In the `mod_jk` source code tree, change directory into the native source code directory corresponding to your version of the Apache server (for example, when using Apache 2.2.3, change directory to `C:\apachesrc\mod_jk\tomcat-connectors-1.2.19-src\native\apache-2.0`). From this directory, execute the command `nmake -f Makefile.vc`.

9. If you get build errors, you may need to edit the `Makefile.vc` makefile to get everything compiled. With Apache 2.2.3, it was necessary to add the paths to the `apr` and the `apr-util` includes, which had been moved to `%APACHE2_HOME%\srclib\apr\include` and `%APACHE2_HOME%\srclib\apr-util\include` respectively (change the `$(CPP_PROJ)` build variable in the makefile for this). During linking, `libhttpd.lib`, `apr.lib`, and `apr-util.lib` are found in `%APACHE2_HOME%\Release\libhttpd.lib`, `%APACHE2_HOME%\srclib\apr\Release\apr-1.lib`, and `%APACHE2_HOME%\srclib\apr-util\Release\apr-util.lib` respectively (change the `$(LINK32_FLAGS)` build variable in the makefile).

10. You will find the compiled `mod_jk.so` in the `Release` subdirectory after successful compilation.

Building mod_jk on Linux/*nix

You need to make sure you have the required `gcc` compiler and `libc` versions on your Linux system to compile the Apache Web server and `mod_jk`. The specific version required is dependent on the Apache Web server that you will be using. In addition, the script used by `mod_jk` installation requires installation of `libtool 1.5.2` or higher, and at least `autoconf 2.59`.

1. Download the source of the Apache Web server, corresponding to the version of the server you are using, from `http://httpd.apache.org/download.cgi`.

2. Unpack the distribution into any convenient directory, such as `/home/dev/httpd-2.2.3`.

3. Download the latest Connector source from `http://tomcat.apache.org/download-connectors.cgi`.

4. Unpack it to a convenient directory, such as `/home/dev/tomcat-connects-1.2.19-src`.

5. Compile and install the Apache server. You should follow the detailed instructions in the `install` file of your Apache server source directory. With the 2.2.3 server, you need to issue the command `./configure --prefix=/installdir` where you can specify the location (`installdir`) where you want the server to be installed. After configuration finishes, type the **make** command to compile the source. Finally, type the **make install** command to install the server.

6. This step compiles the `mod_jk` module as a dynamically loadable module. Change into the native subdirectory. Then enter the command **./configure --with-apxs=/installdir/bin/apxs**. The `installdir` must be the same as the installation directory of your Web server in step 5; be aware that this is not the same as the source code directory for the Web server. This creates a makefile. You can then type **make** to compile the `mod_jk` module. You can then find the `mod_jk.so` module in the apache-2.0 subdirectory.

Instead of creating a dynamically loaded `mod_jk` *module for Apache Web server, it is also possible to statically link* `mod_jk` *into the Apache Web server when you recompile the server. Statically linking* `mod_jk` *into the Web server executable is out of the scope of this book; please consult* `mod_jk` *documentation if you need to perform this task.*

Adding Directives to Load the mod_jk Module (httpd.conf)

Once you have the binaries of the `mod_jk` module, you are ready to modify the Apache Web server's configuration file to load the module.

First, you must make sure you have copied the `mod_jk.so` binary module to the `modules` subdirectory of the Apache server installation.

To load the `mod_jk.so` as a module, edit the `httpd.conf` file by adding an entry, as shown here in the `LoadModules` section.

For Windows, use the following:

```
# For Windows include the actual mod_jk path in double quotes
# if the path contains any white spaces.
LoadModule jk_module modules/mod_jk.so
```

Restart the Apache server, and the module should be loaded.

If you are running on Apache 2.2.*x*, you can use the `httpd -D DUMP_MODULES` command to verify that `mod_jk` is successfully loaded (not available for 1.3.*x* or 2.0.*x*):

```
C:> httpd -D DUMP_MODULES
Loaded Modules:
 core_module (static)
 win32_module (static)
 mpm_winnt_module (static)
 http_module (static)
 so_module (static)
 actions_module (shared)
 alias_module (shared)
 asis_module (shared)
 auth_basic_module (shared)
 authn_default_module (shared)
 authn_file_module (shared)
 authz_default_module (shared)
 authz_groupfile_module (shared)
 authz_host_module (shared)
 authz_user_module (shared)
```

(continued)

```
 autoindex_module (shared)
 cgi_module (shared)
 dir_module (shared)
 env_module (shared)
 imagemap_module (shared)
 include_module (shared)
 isapi_module (shared)
 log_config_module (shared)
 mime_module (shared)
 negotiation_module (shared)
 setenvif_module (shared)
 userdir_module (shared)
 jk_module (shared)
Syntax OK
```

Additional mod_jk Directives

The `mod_jk` module requires additional directives, placed in the `httpd.conf` file, to configure its operation. Remember that the Apache server must be restarted for any changes to `httpd.conf` to take effect.

To tell `mod_jk` where to find the `workers.properties` file, use the `JkWorkersFile` directive:

```
JkWorkersFile     conf/workers.properties
```

The `JkLogFile` directive tells `mod_jk` where to write its logs. For example:

```
JkLogFile     /opt/logs/httpd/mod_jk.log
```

The level of logging (info, error, debug) can be controlled using the `JkLogLevel` directive. The info level will provide a normal level of logging, while error and debug level will provide more logging details. For example, use this to get maximal logging:

```
JkLogLevel  debug
```

You can also ask `mod_jk` to log information on its requests by using the `JkRequestLogFormat` directive (inside the corresponding Apache server `<VirtualHost>` definition if you want to control logging per virtual host). You can use any of the format specifiers shown in the following table.

Format Specifier	Description
`%b` or `%B`	Bytes transmitted, not counting HTTP headers (two different formats)
`%H`	Request protocol
`%m`	Request method
`%p`	Port of server for the request
`%q`	The query string beginning with "?"
`%r`	First line of request
`%T`	Request duration <seconds>.<microseconds>

Format Specifier	Description
%U	URL of request, with query string removed
%v or %V	Server name
%w	Name of Tomcat worker
%R	The route name of the session

For example, to log the Tomcat worker name, URL, and duration for every request, use the following directive:

```
JkRequestLogFormat  "%w  %U  %T"
```

To control the URL matching and forwarding to the Tomcat workers, you need to add `JkMount` directives. The general format of the `JkMount` directive is:

```
JkMount <URL to match>  <Tomcat worker name>
```

For example, to forward all requests destined for `/jsp-examples/` to `worker1`, use the `JkMount` directive:

```
JkMount /examples/jsp/*  worker1
```

To exclude some URLs from being forwarded by `mod_jk`, you can set the `no-jk` environment variable using the `SetEnvIf` directive (you need `mod_setenvif` installed on your Apache Web server for this to work). For example, to exclude all URLs with `/nomap` from `mod_jk` forwarding, you can use the following directive:

```
SetEnvIf Request_URL "/nomap/*" no-jk
```

Testing the mod_jk Setup

This section tests a working Apache server to a Tomcat worker setup, connected via AJP 1.3 and `mod_jk`. Assuming you have `mod_jk` compiled, and have confirmed installation in the Apache Web server, you need to add the following to your `httpd.conf`:

```
JkWorkersFile  conf/workers.properties
JkMount /examples/jsp/*  worker1
```

In the `conf` subdirectory of your Web server installation, place a `workers.properties` file that contains the following:

```
worker.list = worker1
worker.worker1.type = ajp13
worker.worker1.host = 192.168.23.4
```

The host must be changed to the IP address of your own Tomcat worker server; in this case, it is the IP address of the local machine.

On the Tomcat host, make sure you have an AJP connector at the default 8003 port. The default `server.xml` already contains this:

```
<!-- Define an AJP 1.3 Connector on port 8009 -->
 <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

Before testing this setup, restart Tomcat and then restart Apache. This ensures that the configuration changes made for Tomcat and Apache have been read.

To test the setup, point your browser to the following URL and browse to sample JSPs bundled with Tomcat:

```
http://localhost/examples/jsp/
```

Note that you are browsing to `http://localhost/` (the host/port on which Apache is listening) and not `http://localhost:8080/` (Tomcat's host/port). If everything was configured properly, the Web page shown in Figure 11-2 should be displayed.

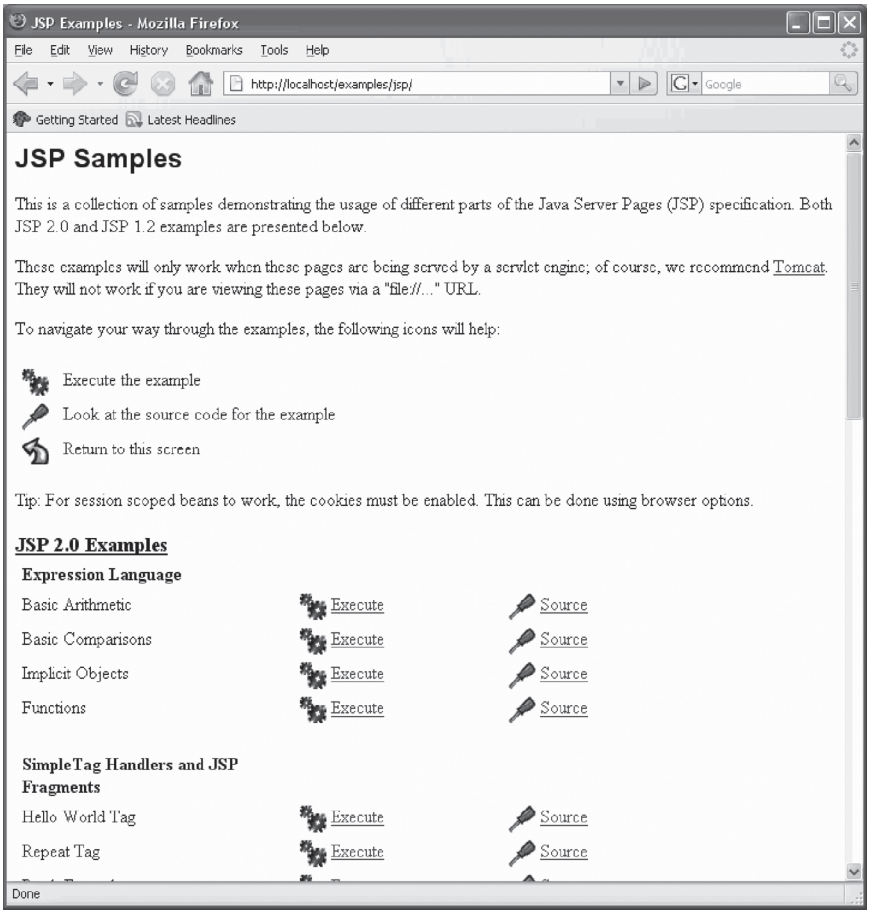
JSP Examples - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/examples/jsp/
Getting Started Latest Headlines
JSP Samples
This is a collection of samples demonstrating the usage of different parts of the Java Server Pages (JSP) specification. Both JSP 2.0 and JSP 1.2 examples are presented below.
These examples will only work when these pages are being served by a servlet engine; of course, we recommend Tomcat. They will not work if you are viewing these pages via a "file://..." URL.
To navigate your way through the examples, the following icons will help:
Execute the example
Look at the source code for the example
Return to this screen
Tip: For session scoped beans to work, the cookies must be enabled. This can be done using browser options.
JSP 2.0 Examples
Expression Language
Basic Arithmetic Execute Source
Basic Comparisons Execute Source
Implicit Objects Execute Source
Functions Execute Source
SimpleTag Handlers and JSP Fragments
Hello World Tag Execute Source
Repeat Tag Execute Source
Done

Figure 11-2: Proper configuration should result in this JSP Samples page

Now try executing one of the JSPs. Click the Date example; this points the browser to `http://localhost/jsp-examples/jsp/dates/date.jsp`.

This should display the Web page shown in Figure 11-3.

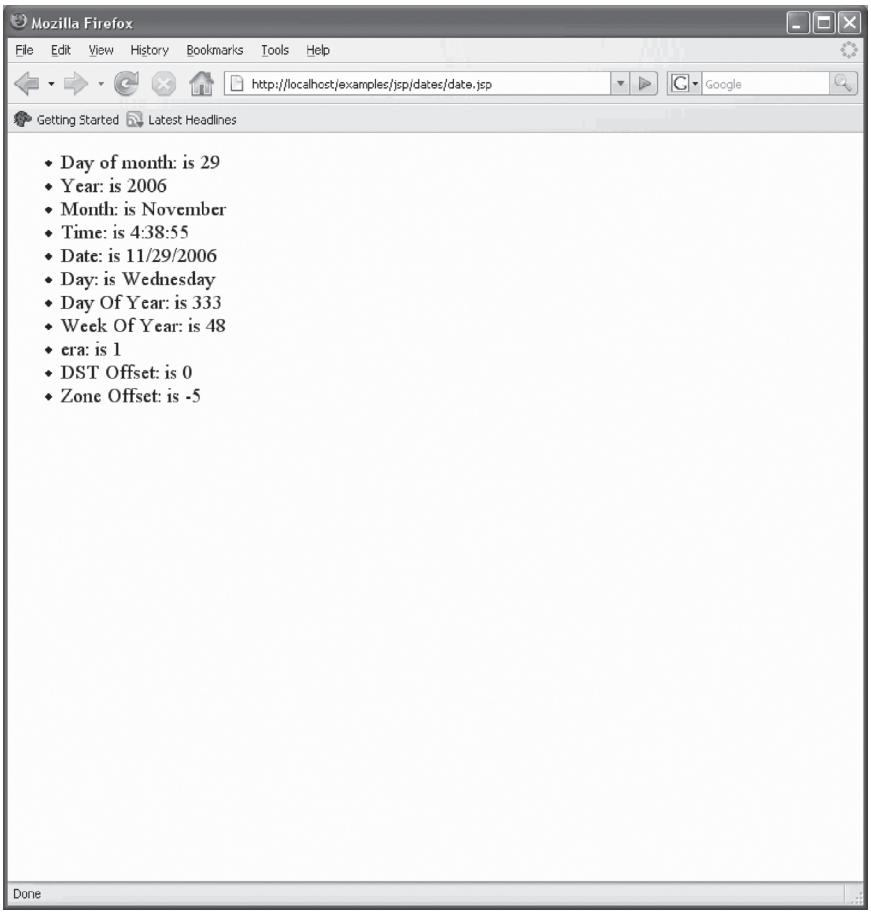

Figure 11-3: Another test of executing a JSP

This test confirms that requests for JSPs are being redirected by Apache to Tomcat correctly. After testing the deployment from a local machine, test the installation from any other machine across the network.

Using the mod_proxy Module

Administrators running Apache 2.2.*x* have a new choice for native modules to use for re-directing requests to Tomcat server instance(s). It is called `mod_proxy`, and the following section shows where to obtain this module, and how to configure it.

Configuring the mod_proxy Module on Apache 2.2.x Server for the AJP Protocol

Starting with release of Apache Web Server 2.2.*x*, much of the ongoing support for working with the AJP protocol module has been transferred to the developers of the `mod_proxy` module. The `mod_proxy` module now supports the AJP 1.3 protocol and is the preferred way of integrating Apache Web Server 2.2.*x* with instances of Tomcat 6 servers. Remember that you should use either `mod_proxy` or `mod_jk`, but not both on the same server installation.

Installing mod_proxy on Windows/Linux

The good news about `mod_proxy` is that you typically do not have to build it separately, and installation is really simple. This is because `mod_proxy` is a standard module that is built with Apache 2.2.*x* whenever you build from source. While it is not installed by default — in order to keep the default server configuration as lean and mean as possible — it is already compiled in binary, and ready for you to add to the configuration.

You can see which modules are loaded and/or compiled for your server by issuing the following command:

```
C:\>httpd -D DUMP_MODULES
Loaded Modules:
 core_module (static)
 win32_module (static)
 mpm_winnt_module (static)
 http_module (static)
 so_module (static)
 actions_module (shared)
 alias_module (shared)
 asis_module (shared)
 auth_basic_module (shared)
 authn_default_module (shared)
 authn_file_module (shared)
 authz_default_module (shared)
 authz_groupfile_module (shared)
 authz_host_module (shared)
 authz_user_module (shared)
 autoindex_module (shared)
 cgi_module (shared)
 dir_module (shared)
 env_module (shared)
 imagemap_module (shared)
 include_module (shared)
 isapi_module (shared)
 log_config_module (shared)
 mime_module (shared)
 negotiation_module (shared)
 setenvif_module (shared)
 userdir_module (shared)
```

To add `mod_proxy` support for your server, edit the `httpd.conf` file and uncomment the following three lines:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
```

After restarting the server, if you try `httpd -D DUMP_MODULES` again, you should see the new modules loaded (highlighted in the following):

```
C:\>httpd -D DUMP_MODULES
Loaded Modules:
 core_module (static)
 win32_module (static)
 mpm_winnt_module (static)
 http_module (static)
 so_module (static)
 actions_module (shared)
 alias_module (shared)
 asis_module (shared)
 auth_basic_module (shared)
 authn_default_module (shared)
 authn_file_module (shared)
 authz_default_module (shared)
 authz_groupfile_module (shared)
 authz_host_module (shared)
 authz_user_module (shared)
 autoindex_module (shared)
 cgi_module (shared)
 dir_module (shared)
 env_module (shared)
 imagemap_module (shared)
 include_module (shared)
 isapi_module (shared)
 log_config_module (shared)
 mime_module (shared)
 proxy_module (shared)
 proxy_ajp_module (shared)
 proxy_balancer_module (shared)
 negotiation_module (shared)
 setenvif_module (shared)
 userdir_module (shared)
Syntax OK
Configuring mod_proxy
```

To tell `mod_proxy` to forward requests for certain URI to the Tomcat server, you need to add the following segment to the `httpd.conf` file. This should be placed in the main section of the configuration, but can also be placed within your Apache Web server virtual host definitions.

```
ProxyRequests Off
ProxyPreserveHost On
<Proxy *>
    Order deny,allow
    Allow from all
</Proxy>
```

(continued)

```
ProxyPass /examples/jsp ajp://192.168.23.228:8009/examples/jsp
ProxyPassReverse /examples/jsp ajp://192.168.23.228:8009/examples/jsp
<Location /examples/jsp >
    Order allow,deny
    Allow from all
</Location>
```

The first directive, `ProxyRequestsOff`, turns off forward proxying. This is essentially turning off the forwarding capabilities of `mod_proxy`, except for the specific mappings that you will be specifying with `ProxyPass` and `ProxyPassReverse` directives.

The `ProxyPreserveHost` directive tells `mod_proxy` to pass the requested host information from the original request to the AJP connection. This is useful for applications that have a dependency on the requested host. This directive is not available for the 1.3.*x* series of Apache Web servers.

The `<Proxy *>` section specifies the access rules using standard Apache configuration syntax. In this case, all incoming hosts can access the proxy. In production, you may want to restrict the set of allowed hosts that can access the proxy. See Apache Web server documentation on the syntax.

The `ProxyPass` directive specifies that requests for the `/examples/jsp` URI should be sent to `localhost:8009/examples/jsp`, and requests should be sent using AJP protocol. Of course, the Tomcat server instance is already set to listen at this port via the Tomcat AJP Connector.

The `ProxyPassReverse` directive is an essential accompanying directive when configuring AJP proxy to Tomcat. This directive specifies that headers of reverse proxy requests should be rewritten appropriately. This ensures that any redirections from the Tomcat server are handled correctly.

The `<Location>` section is a standard Apache Web server section for specifying URI access permissions. In this case, the section ensures that the proxied URI `/examples/jsp` is accessible to all.

Testing the mod_proxy Setup

This section tests a working Apache server to Tomcat server setup, connected via AJP 1.3 and `mod_proxy`. Confirm that you are using Apache 2.2.*x* and that the binary `mod_proxy.so` is located under the `modules` directory.

You need to make sure the following lines are uncommented in your `httpd.conf`.

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
```

Also in the `httpd.conf` file, add the following to the main configuration section (see comments in the Apache Web server's `httpd.conf` file to locate the main section).

```
ProxyRequests Off
ProxyPreserveHost On
<Proxy *>
    Order deny,allow
    Allow from all
</Proxy>
```

```
ProxyPass /examples/jsp ajp://192.168.23.228:8009/examples/jsp
ProxyPassReverse /examples/jsp ajp://192.168.23.228:8009/examples/jsp
<Location /examples/jsp >
    Order allow,deny
    Allow from all
</Location>
```

The host must be changed to the IP address of your own Tomcat worker server; in this case, it is the IP address of the local machine.

On the Tomcat host, make sure you have an AJP connector at the default 8003 port. The default `server.xml` already contains this:

```
      <!-- Define an AJP 1.3 Connector on port 8009 -->
       <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

Before testing this setup, restart Tomcat and then restart Apache. This ensures that the configuration changes made for Tomcat and Apache have been processed.

To test the setup, point your browser to the following URL and browse to sample JSPs bundled with Tomcat:

```
http://localhost/examples/jsp/
```

This assumes that you are running Apache server on the localhost. Note that you are browsing to `http://localhost/` (the host/port on which Apache is listening) and not `http://localhost:8080/` (Tomcat's host/port). If everything was configured properly, the Web page shown in Figure 11-2 should be displayed.

Now try executing one of the JSPs. Click the Date example; this points the browser to `http://localhost/jsp-examples/jsp/dates/date.jsp`.

This should display the Web page shown in Figure 11-3.

This test confirms that requests for JSPs are being redirected by Apache to Tomcat correctly. After testing the deployment from a local machine, you can test the installation from any other machine across the network.

Configuring SSL for Apache Web Server

SSL provides a secure communication channel between the browser and the Web server. When Apache is used with Tomcat, you can use SSL at either the Apache end or the Tomcat end — or both. The preferred option is to enable SSL at the Apache end because of the better SSL support in Apache. Chapter 14 explores how SSL is set up for the HTTP Connector when Tomcat is used in the standalone mode of operation.

This section shows you how to configure SSL for the Apache Web server Tomcat setup. The secured SSL connection can be enjoyed by any Tomcat applications running behind the Apache server, as well as any other non–Tomcat-based application or modules hosted on the same server.

The steps involved to configure SSL support in Apache are listed here:

1. Install OpenSSL on your server, if it is not already installed. Most Linux systems should have OpenSSL installed.
2. Check whether your Apache installation has `mod_ssl` support. If not, you would need to build Apache from source with the `mod_ssl` support.
3. Get or generate an SSL certificate, and install it in Apache.
4. Make configuration changes in Apache for `mod_ssl`.
5. Test the SSL-enabled Apache-Tomcat setup.

These steps are explained in greater detail in the following sections.

> *The versions used in this chapter are Apache 2.2.4 server, with OpenSSL0.9.8d, running on a Linux operating system. Configuration for other versions should be similar; however, you should consult the associated documentation if you need to configure SSL for other version(s) of Apache server, SSL implementation, or operating systems. Also, the location of some of the configuration files may be different if you are using another Linux distribution.*

Configuring mod_ssl for Apache

Apache can be enabled with SSL using the `mod_ssl` Apache module. This section provides an overview of the major steps involved in configuring Apache 2.2.4 with SSL on Linux. A similar setup will work on Windows by changing the appropriate system-specific paths.

As mentioned in the introduction of this chapter, it is assumed that Apache is configured and running on the server.

Verifying OpenSSL Installation

All the popular Linux distributions usually include OpenSSL. You can verify that OpenSSL is installed by typing the following command in a console.

```
openssl version
```

If OpenSSL is installed, you should see a report of the version number, similar to the following.

```
OpenSSL 0.9.8d 28 Sep 2006
```

If you get a command not found error, check the package installation instructions of your Linux distribution and install OpenSSL and the associated development packages, or get it from `www.openssl.org`.

Building Apache with mod_ssl Support from Source

`Mod_ssl` is a standard module that is included with the latest versions of Apache Web servers, and if you have it, you can skip this step. To determine whether `mod_ssl` is included, change directory to `$APACHE_HOME/bin` and run the following command:

```
$ ./ httpd -D DUMP_MODULES
```

Here, `$APACHE_HOME` is the install location of your Apache2 distribution, and this is the `/usr/local/apache2` directory by default on most Linux distributions. Executing this command prints all modules included with your Apache binary; check if this includes `mod_ssl`.

In the unlikely event that you don't already have `mod_ssl` support, you can download and build the binaries yourself. Download the Apache Web server source code from one of the mirrors at the URL:

```
http://httpd.apache.org/download.cgi
```

As mentioned earlier, this example uses Apache 2.2.4, and the download file is `httpd-2.2.4.tar.gz`.

Note that your Linux installation must also have the development packages installed because the compilation depends on the `gcc` *compiler.*

Next, unarchive the Apache server distribution into a working directory:

```
$ tar zxvf httpd-2.2.4.tar.gz
```

Now, change the directory to the source directory and configure the `mod_ssl` module, together with any other modules that you need, using the commands.

```
$ cd httpd-2.2.4$ ./configure --enable-ssl=shared --enable-proxy=shared
--enable-proxy-ajp=shared --enable-proxy-balancer=shared
```

The `shared` value in the command indicates that the module should be built as a DSO loadable library, instead of statically compiled. This command enables the following modules:

- `mod_ssl`
- `mod_proxy`
- `mod_proxy_ajp`
- `mod_proxy_balancer`

In general, to enable any Apache module, use an `--enable-<`*`module name without mod_prefix`*`>` option. If there are any underscores in the module name, replace them with hyphens.

> **By default, the compilation assumes that Apache Web server is installed in** `/usr/local/apache2`**. If you wish to change this, use** `--prefix=/`*`path-to-your-installation-directory`*.

This configuration takes a little while to complete as it checks for dependencies and generates a `make` file. After this configuration, you are ready to compile the source code. Use the command:

```
$ ./make
```

This compilation and linking takes a few minutes on most machines. Finally, you can install the compiled Apache server using the following command:

```
$ ./make install
```

This `make` target installs the application to the default `/usr/local/apache2` directory. This directory location is referred to as `$APACHE_HOME` later in this section. If you have specified an alternative directory using the `--prefix` option during configuration, this installation copies the files to your specified directory instead.

You can now start the server by going to `$APACHE_HOME/bin` and executing the following command:

```
$ ./apachectl start
```

You can stop the server at any time by going to `$APACHE_HOME/bin` and executing the following command:

```
$ ./apachectl stop
```

See the Apache 2 documentation for more information on other commands and options available.

The main Apache configuration file, `httpd.conf`, is located in the `$APACHE_HOME/conf` directory, and you need to edit it to configure SSL.

Generating a Test Certificate with OpenSSL

This section describes the steps required to generate a test certificate for your Apache Web server.

Typically, in a production environment, a commercial-grade certificate from a Certificate Authority (CA) is used. To keep things flowing in this example, you act as your own Certificate Authority by signing the certificate yourself. This, of course, is acceptable only during testing.

Following are the main steps involved:

1. Create a configuration file for generating the certificate.
2. Create a certificate signing request; this is what you submit to a CA if you are buying a certificate.
3. Purchase a certificate from a CA or create a self signed certificate.
4. Remove the passphrase from the private key.
5. Install the key and certificate to the server.

Configuration File for Generating a Certificate

Create a working directory called `certworks`. You can generate all the required requests, keys, configuration, and certificates here.

A configuration file is required for generating the server certificate. A sample configuration file is presented in the following listing. Save the following contents in a file named `myconfig.file` in the `certworks` directory.

```
RANDFILE               = ./random.txt
[req]
default_bits           = 1024
default_keyfile        = keyfile.pem
```

```
attributes              = req_attributes
distinguished_name      = Wiley
prompt                  = no
output_password         = mypassword
[Wiley]
C                       = US
ST                      = NJ
L                       = Hoboken
O                       = Wiley
OU                      = Wrox Press
CN                      = 192.168.23.168
emailAddress            = mail@myserver.com
[req_attributes]
challengePassword       = mypassword
```

If you are testing on your own local LAN, you should change the CN (Common Name) entry to the fully qualified hostname or IP of your host. In the example above, the CN is set to 192.168.23.168.

If you are actually setting this up for a registered fully qualified domain name, this entry *must* match exactly the domain that you are requesting the certificate for. If your users are not using this exact name to access your site, they get a security warning from the browser.

The key generator needs a file containing a random number to add entropy to the algorithm. Create a file called `random.txt` and put a large random number in it.

Create a Certificate Signing Request

The command for creating a certificate signing request is as follows:

```
openssl req -new -out server.csr -config myconfig.file
```

If you use the configuration from the `myconfig.file`, this step creates a certificate signing request (`server.csr`) and a private key (`keyfile.pem`).

The following is a sample output from this command:

```
Generating a 1024 bit RSA private key
..............++++++
..................++++++
writing new private key to 'keyfile.pem'
-----
```

Remove the Passphrase from the Private Key

This is an optional step that should be performed for security reasons. To remove the passphrase from the private key, run the command as shown:

```
openssl rsa -in keyfile.pem -out server.key
```

This command prompts for the password. Use the same password specified in the `myconfig.file` (`mypassword` in this case). The `server.key` should be readable only by the Apache server and the administrator. We highly recommend that you delete the `random.txt` file because it contains the entropy information for creating the key and could be used for cryptographic attacks against your private key.

Create a Self-Signed Certificate

In a production environment, the certificate signing request file generated (`server.csr`) is sent to a Certificate Authority and a certificate purchased.

For test deployments, you can generate a self-signed certificate. The following command shows this being done:

```
openssl x509 -in server.csr -out server.crt -req -signkey server.key -days 365
```

The `-days` option specifies the number of days after which the certificate will expire.

The following is a sample output from this step:

```
Signature ok
subject=/C=US/ST=NJ/L=Hoboken/O=Wiley/OU=Wrox Press/CN=192.168.23.168/emailAddre
ss=mail@myserver.com
Getting Private key
```

The self-signed certificate is generated in the `server.crt` file.

Install the Certificate

Copy the private server key file (`server.key`) and server certificate file (`server.crt`) to the `$APACHE_HOME/conf` directory. Make sure that the `server.key` and `server.crt` can be read by the user running the Apache Web server.

> *Set the file permission of the files in the* `certworks` *directory to protect them from unwanted access (depending on your local policy). You don't need this directory or its files any more because all that Apache requires is the* `server.crt` *and* `server.crt` *files. However, it is useful to keep around — perhaps backed up elsewhere — if you ever need to generate your certificate again.*

Setting Up mod_ssl in Apache

The default SSL configuration file can be found in the `$APACHE_HOME/conf/extra` directory and is called `httpd-ssl.conf`. This file would then need to be included from `httpd.conf`. Edit this file following the extensive comments if you need to customize the configuration.

Some directives you might need to tweak include the following:

- `SSLCertificateKeyFile`: Path to the server private key file (i.e., the `server.key` file)
- `SSLCertificateFile`: Path to the server certificate file (i.e., the `server.crt` file)
- `VirtualHost`: The SSL virtual host context. If you are setting up virtual hosts, or even redirecting to a Tomcat worker, this is the place where you should make your configuration changes. The `DocumentRoot` in the default `VirtualHost` points to Apache's `DocumentRoot` — let this remain unchanged for now.

Finally, you need to make a few edits in the `$APACHE_HOME/conf/httpd.conf` so that Apache can use the `mod_ssl` extension.

First, uncomment or add this line (if not already existing) to load the `mod_ssl` library:

```
LoadModule ssl_module modules/mod_ssl.so
```

Then, find and uncomment the following line, to include the `mod_ssl` configuration file:

```
Include  conf/extra/httpd-ssl.conf
```

Testing the SSL-Enabled Apache Setup

First, test the SSL setup in Apache: Restart Apache using the `apachectl` command and view the HTTPS URL, which is `https://192.168.23.168/` in our example.

You would need to change the IP address to match that of your server — even the port, if you are running on a port other than 80. Also, note the use of `https`, and not `http`. You should now see browser warnings as described in the next section, and then finally the default Apache "It works!" message.

Browser Security Warnings

Because the server certificate is not signed by any well-known CA authority, but is self-signed, you would expect the browser to detect this and warn you when you try to access the page. Indeed, this is what happens.

On Internet Explorer, the browser will pop up with a security dialog box similar to Figure 11-4.

Security Alert
Information you exchange with this site cannot be viewed or changed by others. However, there is a problem with the site's security certificate.
The security certificate was issued by a company you have not chosen to trust. View the certificate to determine whether you want to trust the certifying authority.
The security certificate date is valid.
The security certificate has a valid name matching the name of the page you are trying to view.
Do you want to proceed?
Yes
No
View Certificate

Figure 11-4: Security alert dialog box when using unknown CA on Internet Explorer

If you select View Certificate, Internet Explorer shows you the details of the certificate, as shown in Figure 11-5.

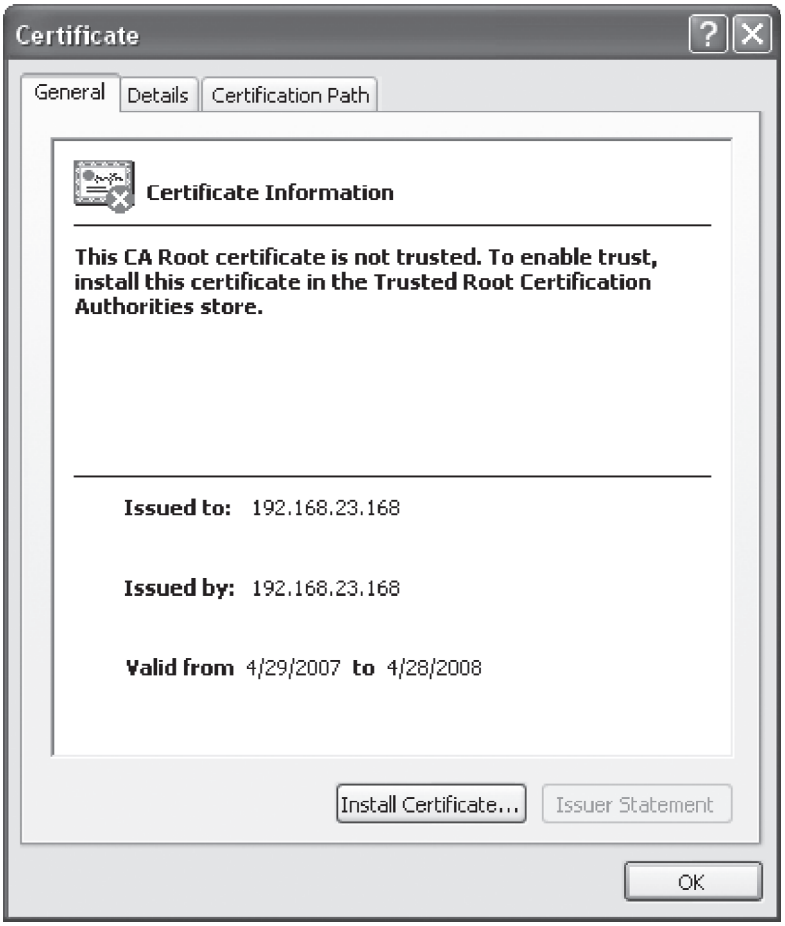

Figure 11-5: Internet Explorer's display of the server certificate

Click OK on this certificate viewing dialog box, and then click Yes on the security warning dialog box to use this certificate for this session. At this time, you should be in an SSL session with the Apache server. Look at the lower-right corner of the Internet Explorer window. On the right side of the status bar you should see the familiar comfort-assuring yellow lock, as shown in Figure 11-6.

Figure 11-6: The Secured Connection Indicator on Internet Explorer

If you are using Firefox, the warning dialog box for an unknown authority is shown in Figure 11-7.

If you click the Examine Certificate button, Firefox displays the certificate in a dialog box, as shown in Figure 11-8.

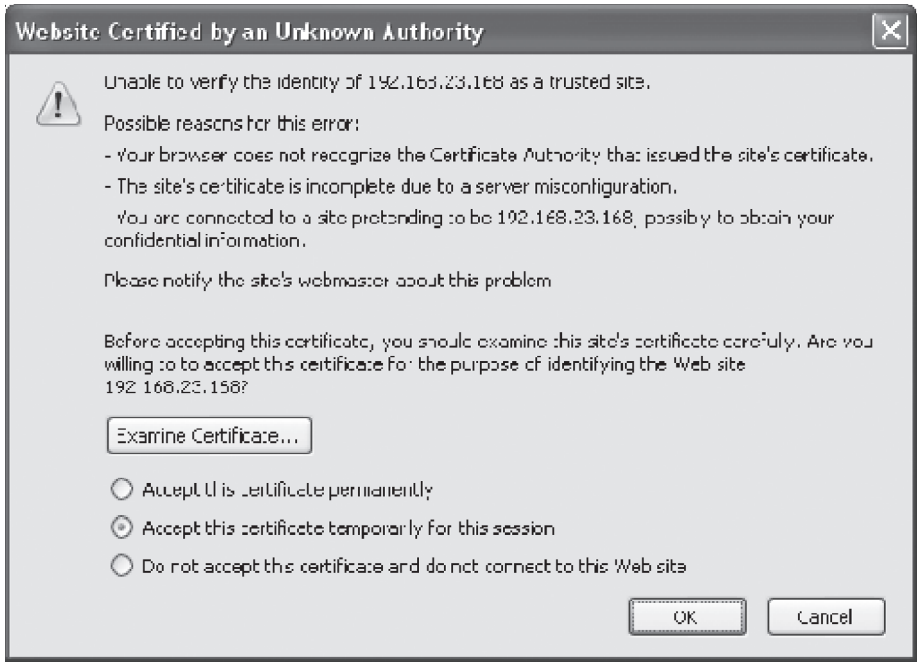
Website Certified by an Unknown Authority
Unable to verify the identity of 192.168.23.168 as a trusted site.
Possible reasons for this error:
- Your browser does not recognize the Certificate Authority that issued the site's certificate.
- The site's certificate is incomplete due to a server misconfiguration.
You are connected to a site pretending to be 192.168.23.168, possibly to obtain your confidential information.
Please notify the site's webmaster about this problem
Before accepting this certificate, you should examine this site's certificate carefully. Are you willing to accept this certificate for the purpose of identifying the Web site 192.168.23.168?
Examine Certificate...
Accept this certificate permanently
Accept this certificate temporarily for this session
Do not accept this certificate and do not connect to this Web site
OK
Cancel

Figure 11-7: Security dialog box when using Unknown CA on Firefox

Click Close in the Certificate Display dialog box, and OK in the Unknown Authority Warning dialog box. This makes Firefox use the certificate for this session only. If you look at the lower-right corner of the Firefox window, you should see the secured connection indicator shown in Figure 11-9.

SSL-Enabled Apache-Tomcat Setup

Now that your have working SSL support in Apache, the next step is to get the JSP/servlet requests sent to Apache passed on to Tomcat. This can be done via either `mod_jk` or `mod_proxy`.

If you are using `mod_jk`, note that you need to have `mod_jk` compiled as a part of your Apache server along with `mod_ssl`. Check if you do, and then follow the steps mentioned in the section "Using the mod_jk Module" earlier in this chapter, such as the configuration changes for the AJP Connector in Tomcat's `server.xml`, and the `mod_jk`-related directives in Apache's `httpd.conf`, such as `LoadModule`, `JkWorkersFile` setting, and so on.

The only change from that configuration is that you need to use the `<VirtualHost>` declared in `httpd-ssl.conf` for port 443, instead of defining one in `httpd.conf`, and place your `JkMount` directives inside it. The following shows an example modification to the `$APACHE_HOME/extra/httpd-ssl.conf` file.

```
<VirtualHost _default_:443>
...

JkWorkersFile  conf/workers.properties
JkMount /examples/jsp/*  worker1

</VirtualHost>
```

Figure 11-8: Firefox dialog box displaying the server certificate

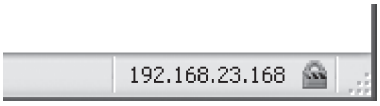

Figure 11-9: The secured connection indicator on Firefox

If you are using `mod_proxy` instead, the section "Building Apache with mod_ssl Support" showed the step for compiling in the `mod_proxy` module along with `mod_ssl`. Now follow the steps outlined in the "Using the mod_proxy Module" section earlier in the chapter, including the following:

- In the `$APACHE_HOME/conf/http.conf` file, make sure that you have `LoadModule` for `mod_proxy`, `mod_proxy_ajp`, and `mod_proxy_balancer`.

- In the `$APACHE_HOME/conf/extra/httpd-ssl.conf` file, add the proxy configuration detailed in the section "Using the mod_proxy Module" into the default `<VirtualHost>` element. The following configuration listing shows this modification. Of course, you need to replace 192.168.23.228 with the IP address of your Tomcat 6 server.

```
<VirtualHost _default_:443>
...

ProxyRequests Off
ProxyPreserveHost On
<Proxy *>
    Order deny,allow
    Allow from all
</Proxy>
ProxyPass /examples/jsp ajp://192.168.23.228:8009/examples/jsp
ProxyPassReverse /examples/jsp ajp://192.168.23.228:8009/examples/jsp
<Location /examples/jsp >
    Order allow,deny
    Allow from all
</Location>

</VirtualHost>
```

  Make sure you have configured an AJP connector on the Tomcat server.

- Finally, restart Tomcat and then restart Apache.

To test the SSL-enabled Apache-Tomcat setup, try the following URL:

```
https://192.168.23.168/examples/jsp/dates/date.jsp
```

You should replace the IP address with the name of your server, and the exact name you specified for the CN element of your server certificate. This should execute the date example bundled with the Tomcat distribution. You should see similar indications confirming that you have a secured HTTPS connection to this JSP. In case of any errors, as before, refer to the logs in the `$APACHE_HOME/logs` directory.

> *If you use SSL at the Apache end only, the connection between Apache and Tomcat may still be unsecured. This may be of concern in deployments where Apache runs on a server in the DMZ (demilitarized zone), and Tomcat is behind an internal firewall. One way of addressing this is by using an SSH (Secure Shell) tunnel to encrypt the AJP data passing between Apache and Tomcat. The details of setting up an SSH tunnel are not covered here, but if you do set up such a tunnel, you should remember to change the host/port values in Apache's* `workers.properties` *file to point to the tunnel, instead of directly to the Tomcat server. Another way is to use the OpenSSL support for the AJP Connector via the APR Connector.*

Tomcat Load Balancing with Apache

In this chapter, we cover only a basic implementation of load balancing. Chapter 18 describes a more sophisticated environment, with support for persistent sessions with in-memory session replication. The configuration described has been tested on Apache 2.2.3 with `mod_jk 1.2.19` on a Windows system.

The `mod_proxy` balancer module can also be used for load balancing; `mod_proxy` balancer is not covered in this chapter.

Enterprise Web applications must be fast, scalable, and reliable, and offer fail-safe behavior. For high-traffic Web sites, it is a good idea to route the requests coming from Apache to multiple Tomcat instances, instead of just one. The `mod_jk` module supports load balancing with `seamless sessions`. It uses a simple round-robin scheduling algorithm. For each Tomcat worker, a weight can be assigned in the `workers.properties` file, which specifies how the request load is distributed between the workers.

A *seamless session* is also known as *session affinity* or a *sticky session*. When a client requests any dynamic resource for the first time, the load balancer will route this request to any of the available Tomcat instances. Any subsequent requests from the same browser session should be routed to the *same* Tomcat Web container to keep the same user session. If the maximum number of connections for a Tomcat worker is reached, then `mod_jk` waits for it until it is free. This behavior is known as a seamless session. The client experiences no break in application functionality because the associated client session is kept intact.

The `mod_jk` module inherently supports load balancing. The Apache Web server needs some configuration and multiple Tomcat instances to enable load balancing. The next section describes in detail how to set up a load balancer. The example setup consists of one Apache server and three Tomcat instances (workers) running on a single machine, although they could very well be distributed across different machines.

The steps involved in setting up Tomcat load balancing are as follows:

1. Change `CATALINA_HOME` in the Tomcat startup files to point to different locations for each of the Tomcat instances.
2. Set different AJP Connector ports for the instances.
3. Set different server ports.
4. Disable the Coyote HTTP/1.1 Connector.
5. Set the `jvmRoute` in the Standalone Engine.
6. Comment out the Catalina Engine.
7. Configure the Tomcat worker in `workers.properties`.

These steps are covered in detail in the following sections. The three Tomcat workers used for load balancing are referred to as `Tomcat6A`, `Tomcat6B`, and `Tomcat6C`. All of them are on the same machine as Apache. Because multiple Tomcat workers are running on the same machine, they will use different ports for listening to AJP requests. Had these instances been running on different physical machines, they could have used the same port.

> *Before configuring Tomcat workers for Apache, make sure you stop all running instances of Apache and Tomcat. Once the configuration is complete, start all Tomcat instances one by one and then start the Apache Web server.*

Changing CATALINA_HOME in the Tomcat Startup Files

The basic requirement for all the Tomcat instances participating in this load-balanced framework is that all of them should be available simultaneously. Each instance needs a separate `CATALINA_HOME` variable at runtime. A different `CATALINA_HOME` variable can be provided for each of the Tomcat workers by editing the `startup.bat` (`startup.sh` on Unix/Linux) script file. The `startup.bat` file actually

contains code that will guess where CATALINA_HOME is when it is not set. You don't need to perform the following modifications if you don't have CATALINA_HOME set globally.

To force a different CATALINA_HOME for each Tomcat 6 instance, modify the startup.bat file for Tomcat6A, as shown here:

```
set CATALINA_HOME=c:\apps\Tomcat6A
```

For Tomcat6B, use this:

```
set CATALINA_HOME=c:\apps\Tomcat6B
```

For Tomcat6C, use this:

```
set CATALINA_HOME=c:\apps\Tomcat6C
```

> *The* CATALINA_HOME *environment variable should not be globally set in a load-balancing environment when you have more than one Tomcat instance running on the same machine. This is because each Tomcat worker needs its own* CATALINA_HOME*, as shown in the previous configuration.*

This step is not required if you run each Tomcat worker on a different machine.

Setting Different AJP Connector Ports

Because all the Tomcat workers (i.e., Tomcat6A, Tomcat6B, and Tomcat6C) are running on the same machine, each is required to listen on a different port to avoid port conflicts. By default, the AJP 1.3 Connector listens on port 8009, which is preconfigured in Tomcat. Use port 8010 for Tomcat6B, and port 8011 for Tomcat6C.

To configure the AJP Connector for a Tomcat instance, edit the server.xml file and set the AJP port for each of them as explained earlier. Edit the information for the <Connector> tag in this file with appropriate values for the current Tomcat instance:

```
        <!-- Define an AJP 1.3 Connector on port 8009 -->
        <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

Modify the AJP Connector port for each Tomcat worker as shown here:

- For Tomcat6A, use 8009.
- For Tomcat6B, use 8010.
- For Tomcat6C, use 8011.

This step is not required if you run each Tomcat worker on a different machine.

Setting Different Server Ports

To avoid startup port conflicts, edit server.xml and set each Tomcat worker's server port. Locate and modify the following entry for each Tomcat worker:

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
```

For each Tomcat worker set the port as follows:

- For Tomcat6A, use 8005.
- For Tomcat6B, use 8006.
- For Tomcat6C, use 8007.

This step is not required if you run each Tomcat worker on a different machine.

Disabling the Default HTTP/1.1 Connector

Because all the Tomcat instances will be running in conjunction with the load-balancer worker, it's possible that someone could directly access any of the available workers via the default HTTP Connector, bypassing the load-balancer path. To avoid this, comment out the HTTP Connector configuration of all the Tomcat instances in the `server.xml` file, as shown here:

```
<!-- Define
<Connector port="8080" protocol="HTTP/1.1"
           maxThreads="150" connectionTimeout="20000"
           redirectPort="8443" />
-->
```

Setting the jvmRoute in the Standalone Engine

An important step for load balancing is specifying the `jvmRoute`. Each Tomcat worker has an `Engine` directive in the `server.xml` file. The Engine is a top-level container in the Catalina hierarchy and represents the entire Catalina Servlet Engine. This `Engine` directive has an attribute called `jvmRoute` that acts as an identifier for that particular Tomcat worker. Typically, a unique string is provided as the value for this attribute. This string must be unique across all the available Tomcat instances participating in the load-balancing environment.

Add a unique `jvmRoute` attribute to each Tomcat worker's `server.xml` file as described here. This unique `jvmRoute` ID is used in the `workers2.properties` file for identifying each Tomcat worker. Ensure that the strings used are unique for each Tomcat worker. For the configuration discussed here, use the following entries:

For `Tomcat6A` on the `localhost` machine, the entry will be as follows:

```
<!-- You should set jvmRoute to support load-balancing via AJP -->
<Engine name="Standalone" defaultHost="localhost" jvmRoute="Tomcat6A">
```

For Tomcat6B on the `localhost` machine, the entry would look like this:

```
<!-- You should set jvmRoute to support load-balancing via AJP -->
<Engine name="Standalone" defaultHost="localhost" jvmRoute="Tomcat6B">
```

For Tomcat6C on the `localhost` machine, the entry would look like this:

```
<!-- You should set jvmRoute to support load-balancing via AJP -->
<Engine name="Standalone" defaultHost="localhost" jvmRoute="Tomcat6C">
```

Commenting Out the Catalina Engine

After adding the `Engine` directive as shown earlier, you are left with two entries for the `Engine` directive in your `server.xml` file(s). The first is the `Standalone` Engine and the second is the `Catalina` Engine. You need to comment out the `Catalina` Engine directive for each of the Tomcat workers, as shown here:

```
<!-- Define the top level container in our container hierarchy
    <Engine name="Catalina" defaultHost="localhost" >
-->
```

Directives in httpd.conf

Let the `mod_jk` know where the file to configure the workers is located. Use the `JkWorkersFile` directive. Add this directive to the Apache Web server's `httpd.conf`.

```
JkWorkersFile  conf/workers.properties
```

A `JkMount` directive needs to be added to the Apache Web server's `httpd.conf` to redirect the JSP example URLs to the load-balancing worker.

```
JkMount /examples/jsp/*  bal1
```

The load-balancing worker is named `bal1`. The configuration of this worker is discussed in the next section.

Add another `JkMount` directive to map requests for the `jkstatus` URL to the balance worker.

```
JkMount /jkstatus/       stat1
```

The status worker is named `stat1`, and is configured in the next section.

Workers Configuration in workers.properties

This section explains how to specify the properties of the Tomcat workers in the `workers.properties` file.

Configuring Tomcat Worker Instances

A separate worker must be specified for each available Tomcat worker. The following segment from the `workers.properties` file shows how this is done.

For Tomcat6A, the `worker.properties` file settings are as follows:

```
worker.Tomcat6A.type = ajp13
worker.Tomcat6A.host = 192.168.23.4
worker.Tomcat6A.port = 8009
worker.Tomcat6A.lbfactor = 10
```

For Tomcat6B, the `worker.properties` file settings are as follows:

```
worker.Tomcat6B.type = ajp13
worker.Tomcat6B.host = 192.168.23.4
worker.Tomcat6B.port = 8010
worker.Tomcat6B.lbfactor = 10
```

For Tomcat6C, the `worker.properties` file settings are as follows:

```
worker.Tomcat6C.type = ajp13
worker.Tomcat6C.host = 192.168.23.4
worker.Tomcat6C.port = 8011
worker.Tomcat6C.lbfactor = 10
```

You would need to change the host and port values to the actual ones in your deployment. Note that the `lbfactor` for each of the Tomcat workers is configured to the same value. This causes the load balancer, configured in the next section, to share the incoming request load equally across the three Tomcat worker instances.

Configuring Load Balancer in workers.properties

This section discusses how to set up the `workers.properties` file for load balancing. First, you need to create a load-balance worker. The following line in `workers.properties` creates this specialized worker, called `bal1`.

```
worker.bal1.type = lb
worker.bal1.sticky_session = 1
```

Note that the `sticky_session` attribute is set to `1`. This is also the default value, which means that the same Tomcat server instance will be used to service the requests from the same session if possible. You should disable this only if you are using a specialized session manager that can maintain persistent state between multiple physical Tomcat servers (such as physical clustering controller hardware).

To tell `mod_jk` the set of workers that the load balancer should balance across, use the `balance_workers` attribute:

```
worker.bal1.balance_workers = Tomcat6A, Tomcat6B, Tomcat6C
```

Configuring a Status Worker in workers.properties

The status worker is used to display load-balancing statistics in real time. The following segment configures a status worker, named `stat1`, in the `worker.properties` file:

```
worker.stat1.type = status
```

The only attribute available for the status worker is `css`. Use this if you want to specify a stylesheet to format the display of the status worker. The default one, shown in Figure 11-5, is actually quite usable, and the `css` attribute is seldom set.

Supplying mod_jk with a workers.list

Now that the Tomcat worker instances, the load-balancing worker that manages them, and the status workers are all defined, you can tell `mod_jk` about the list of workers. This line in `workers.properties` establishes the list of workers that `mod_jk` can send requests to:

```
worker.list = bal1,stat1
```

Any request forwarded to `bal1` is distributed to one of the managed Tomcat6A, Tomcat6B, or Tomcat6C worker instances.

The Complete workers.properties File

The following is the completed `workers.properties` file with all the elements previously discussed added:

```
worker.list = bal1,stat1
worker.Tomcat6A.type = ajp13
worker.Tomcat6A.host = 192.168.23.4
worker.Tomcat6A.port = 8009
worker.Tomcat6A.lbfactor = 10
worker.Tomcat6B.type = ajp13
worker.Tomcat6B.host = 192.168.23.4
worker.Tomcat6B.port = 8010
worker.Tomcat6B.lbfactor = 10
worker.Tomcat6C.type = ajp13
worker.Tomcat6C.host = 192.168.23.4
worker.Tomcat6C.port = 8011
worker.Tomcat6C.lbfactor = 10
worker.bal1.type = lb
worker.bal1.sticky_session = 1
worker.bal1.balance_workers = Tomcat6A, Tomcat6B, Tomcat6C
worker.stat1.type = status
```

Again, you would need to change the host and port values to the actual ones in your deployment. The load balancer will use a default weighted (by `lbfactor`), request based on the round-robin algorithm for load balancing and will support seamless sessions as discussed earlier. The algorithm is modifiable via the `method` property as described in the earlier section "Other Worker Properties." If a worker dies, the balanced worker will check its state over the configured time intervals of recovery. Until it is back online, all work is redirected to the other available workers.

Testing the Load Balancer

This section explains how to test the load-balancing setup that was configured in the previous sections. To do this, create three similar JSPs for each of the Tomcat workers (using the same filename) and place them into the `webapps/examples/jsp/` directory of each Tomcat worker. Create a file `index.jsp` with the following contents:

```
<%@ page language="java" %>
<html>
  <body>
    <h1><font color="red">Index Page Served By Tomcat6A</font></h1>
    <table align="centre" border="1">
      <tr>
        <td>Session ID</td>
        <td><%= session.getId() %></td>
      </tr>
      <tr>
        <td>Created on</td>
        <td><%= session.getCreationTime() %></td>
      </tr>
    </table>
  </body>
</html>
```

Make the following change to the `index.jsp` file of these copied versions (to identify that the corresponding Tomcat instance has processed the request).

For Tomcat6A, change the following line in the `index.jsp` file:

```
        <h1><font color="red">Index Page Served By Tomcat6A</font></h1>
```

For Tomcat6B, change this line:

```
        <h1><font color="blue">Index Page Served By Tomcat6B</font></h1>
```

For Tomcat6C, the line will be as follows:

```
    <h1><font color="green">Index Page Served By Tomcat6C</font></h1>
```

Make sure that all the Tomcat instances (Tomcat6A, Tomcat6B, and Tomcat6C) are up and running properly.

Restart the Apache Web server to make sure it reads the latest configuration files.

You are now ready to test load balancing. Make sure Apache is serving the static pages. This can be tested by visiting the following URL:

```
http://localhost/
```

Apache will return the `index.htm` page, which just shows "It works!" with the 2.2.*x* version. Now, visit the following URL to confirm that Apache is serving dynamic requests:

```
http://localhost/examples/jsp/index.jsp
```

The request can be served by any of the three Tomcat instances. If Tomcat6A has served the index JSP, then the response would be something like the one shown in Figure 11-10. Depending on which Tomcat worker gets the request, the `index.jsp` page will be served.

Testing Sticky Sessions

To test whether the same Tomcat worker maintains the session, make a note of the session ID for each request. In this case, the corresponding Tomcat worker name is appended to the session ID. In Figure 11-4, `Tomcat6A` is the unique string appended to the session ID. This confirms that `Tomcat6A` has served the request. The Tomcat worker name is determined by the `jvmRoute` attribute, which is set in the `server.xml` file. Whether the same Tomcat worker (which has served the first request) maintains a session can be confirmed by refreshing the browser window repeatedly. You can confirm that the session ID always remains the same for a given Tomcat instance. Hence, the session information remains intact.

If more browser instances are opened, then, for each of them, the request is served by one of the Tomcat instances based on the `lb_factor`. Refreshing each browser window confirms that the sticky sessions are supported by other Tomcat instances as well.

> *Note that the latest version of Firefox actually shares session information between active windows. We recommend that you use Internet Explorer for this test. If you are using Firefox, make sure you have closed all windows and that you restart the browser each time.*

If you notice that load balancing with sticky session support is not working properly, then check whether you have properly configured the `jvmRoute` *attribute of the Standalone Engine directive in* `server.xml` *and commented out the* `Catalina Engine` *directive in all the* `server.xml` *files of the Tomcat workers.*

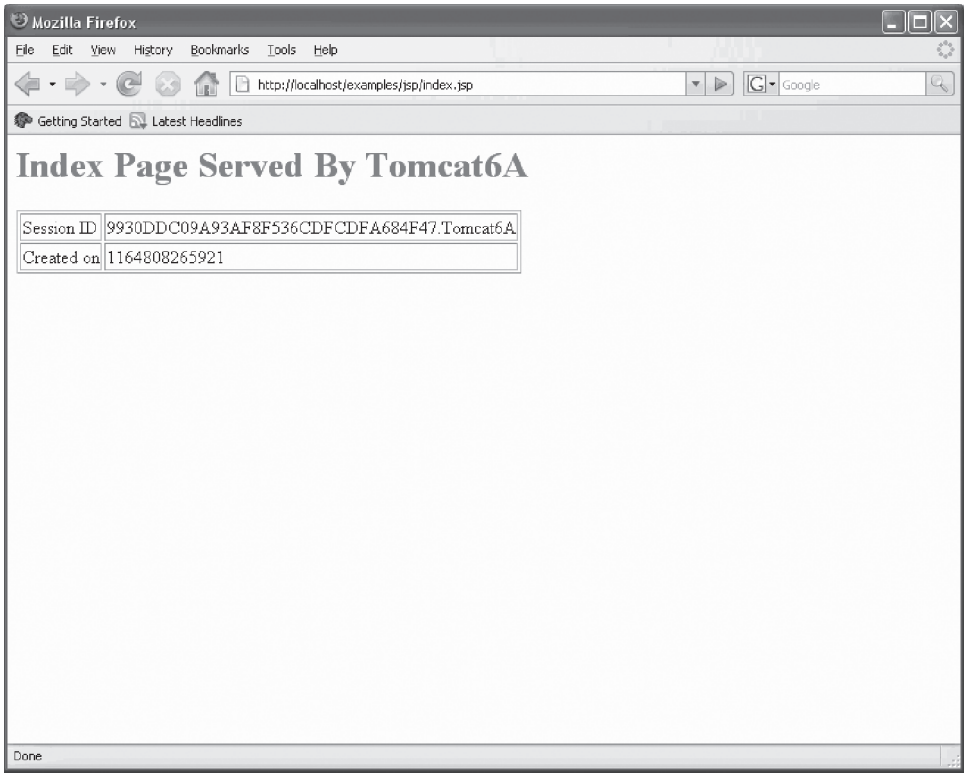

Figure 11-10: Testing the load-balancing behavior

Testing Round-Robin Behavior

The `mod_jk` module implements a round-robin algorithm. To test this, open a few browser windows and visit the following page:

```
http://localhost/examples/jsp/index.jsp
```

You will notice that with different requests from different browsers, different Tomcat workers will serve the request.

The runtime behavior of the Tomcat workers can be tested using the `jkstatus` Web page. The `jkstatus` page keeps track of the status of each worker and maintains a record of the traffic to each Tomcat worker. Apart from this, you can gather a lot of other useful information about the performance of each worker. To view the status, visit the following URL (see Figure 11-11):

```
http://localhost/jkstatus/
```

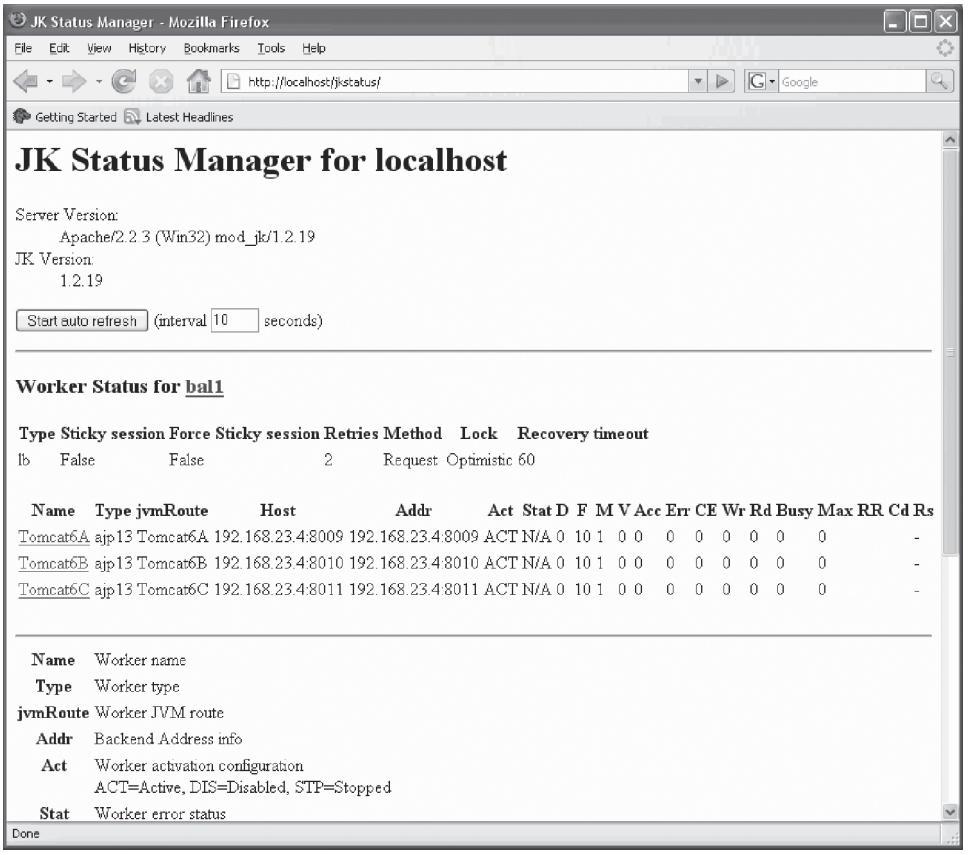
JK Status Manager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/jkstatus/
Getting Started Latest Headlines
JK Status Manager for localhost
Server Version:
Apache/2.2.3 (Win32) mod_jk/1.2.19
JK Version:
1.2.19
Start auto refresh (interval 10 seconds)
Worker Status for bal1
Type Sticky session Force Sticky session Retries Method Lock Recovery timeout
lb False False 2 Request Optimistic 60
Name Type jvmRoute Host Addr Act Stat D F M V Acc Err CE Wr Rd Busy Max RR Cd Rs
Tomcat6A ajp13 Tomcat6A 192.168.23.4:8009 192.168.23.4:8009 ACT N/A 0 10 1 0 0 0 0 0 0 0 0 0 -
Tomcat6B ajp13 Tomcat6B 192.168.23.4:8010 192.168.23.4:8010 ACT N/A 0 10 1 0 0 0 0 0 0 0 0 0 -
Tomcat6C ajp13 Tomcat6C 192.168.23.4:8011 192.168.23.4:8011 ACT N/A 0 10 1 0 0 0 0 0 0 0 0 0 -
Name Worker name
Type Worker type
jvmRoute Worker JVM route
Addr Backend Address info
Act Worker activation configuration
ACT=Active, DIS=Disabled, STP=Stopped
Stat Worker error status
Done

Figure 11-11: The jkstatus Web page

Now, taking this to the next level, check whether Tomcat workers will serve the incoming requests in a round-robin fashion. Keep the browser windows open and you will notice that all three Tomcat workers do serve the incoming requests in a round-robin fashion. This means that if the first request is served by Tomcat6A, Tomcat6B serves the second, and Tomcat6C serves the third. Tomcat6A again serves the fourth request. This is shown in the following table.

Tomcat6A	Tomcat6B	Tomcat6C
1	2	3
4	5	6
7	8	9

Now, stop the Tomcat6B worker and try the same thing. This time, Apache uses the round-robin rule for the remaining two Tomcat workers. The following table reflects the modified request processing.

Tomcat6A	Tomcat6B	Tomcat6C
1	X	2
3	X	4
5	X	6

What happens if Tomcat6B is started again? Does the load balancer realize that Tomcat6B is again available? Moreover, when will the load balancer start using it? The answer is that the load balancer will start using Tomcat6B as soon as it finds that the server is up. It periodically checks the status of the worker, and will start using it as soon as it is made available. This lookup period is equal to the value set for the `recovery` property of the load-balanced worker (see Figure 11-5). The default value is 60 seconds. This can be cross-checked by starting Tomcat6B again and continuing the testing cycle. The response will be something like the following:

Tomcat6A	Tomcat6B	Tomcat6C
1	X	2
3	X	4
5	6	7
8	9	10

Testing with Different Load Factors

In some deployment scenarios, the hardware configurations of all the machines may not be the same. In addition, there is a good chance that even though the hardware configurations are the same, the machines may be serving different online content. Therefore, every machine may not be in a position to contribute exactly the same resources as the others in the final load-balancing setup. This can be handled by adjusting the relative values of the load-balance factor (`lbfactor`) for each of the Tomcat worker instances. The runtime load balancer will distribute the request load appropriately.

To carry out the test, change the `lbfactor` properties in the `workers.properties` file.

For Tomcat6B, make this modification:

```
# Tomcat6B
worker.Tomcat6B.type = ajp13
worker.Tomcat6B.host = 192.168.23.4
worker.Tomcat6B.port = 8010
worker.Tomcat6B.lbfactor = 5
```

For Tomcat6C, make this modification:

```
# Tomcat6C
worker.Tomcat6C.type = ajp13
worker.Tomcat6C.host = 192.168.23.4
worker.Tomcat6C.port = 8011
worker.Tomcat6C.lbfactor = 5
```

Now, restart all three Tomcat workers and then restart Apache. Perform the same test you used to check the load balancing. Browse to the following URL a few times and notice the behavior:

```
http://localhost/examples/jsp/index.jsp
```

You will notice in this case that the behavior has changed. This is because the `lbfactor` setting is causing the load-balancing worker to distribute the request load proportionally; your results should be similar to the set shown in the following table.

Tomcat6A	Tomcat6B	Tomcat6C
1	X	X
2	X	X
X	3	X
X	X	4
5	X	X
6	X	X
X	7	X
X	X	8

Summary

This chapter presented details about the front-ending of one or more Tomcat servers with the Apache Web server. This requires a binary code module, either `mod_jk` or `mod_proxy`, at the Apache server end; and an AJP Connector at the Tomcat 6 end. The topics covered include the following:

- An overview of how Apache Web server can connect to Tomcat 6 instances via the AJP 1.3 protocol
- Getting and building `mod_jk` binary module for the Apache Web server
- The `mod_jk` module and different versions of Apache Web server
- Building and configuration of `mod_proxy` for Apache Web Server 2.2.*x*
- Configuration of `mod_jk` and the AJP Connector for connecting Tomcat with Apache
- The different types of Tomcat workers
- Configuring SSL for an Apache 2.2.*x* Web server
- Tomcat load balancing, and testing load-balancing configurations

Chapter 12 discusses connecting Tomcat with an IIS frontend.

12

Tomcat and IIS

The previous chapter discussed how Apache could be used as a front end to Tomcat. This chapter details the use of Internet Information Services (IIS) with Tomcat. IIS is a popular Web server for Web sites hosted on Microsoft platforms and is used for deploying server-side solutions developed in ASP, C#, and other Microsoft technologies. The primary reason for running IIS along with Tomcat is to allow service providers to support heterogeneous server-side solutions (for example, both ASPs and JSPs) on the same platform.

The Tomcat project provides an Internet Services Application Programming Interface (ISAPI) redirector plug-in for connecting IIS and Tomcat. This chapter covers installation and configuration of the ISAPI plug-in to connect Tomcat with IIS, including the following topics:

- Role of the ISAPI plug-in
- Configuring Tomcat and IIS to work together
- Suggested architectures offering greater scalability
- Troubleshooting tips

The configuration described in this chapter has been tested with IIS 5, which is the version supported for Windows XP Professional and Windows 2000 Server. For deployments on IIS 6 (i.e., on Windows Server 2003) you need to switch IIS to the *IIS 5 isolation mode*, as described later in the chapter.

Role of the ISAPI Plug-in

The integration of Tomcat with a Web server is accomplished with the help of two components: an IIS Web server–specific component and a Java-based Connector implementation. These two components communicate with each other using the Apache JServ Protocol (AJP).

The IIS Web server–specific component is implemented as an ISAPI plug-in. *ISAPI*, or Internet Server Application Programming Interface, is an API that allows developers to write applications

for IIS. ISAPI allows for two kinds of applications: extensions and filters. Extensions are IIS applications, and can be accessed by end users just as an HTML page is accessed. Filters, as the name suggests, filter incoming requests or outgoing responses. Both of these applications are implemented as Dynamic Link Libraries (DLL).

Tomcat provides an ISAPI plug-in called `isapi_redirect.dll` that is used by IIS to serve requests for JSP and servlets. The ISAPI plug-in works *both* as a filter as well as an IIS Web application extension.

As a filter, it watches all IIS requests for those that match specific patterns — such as for JSPs or servlets — and redirects them to Tomcat using the extension. Tomcat then handles these requests and sends the response back to the client via an ISAPI extension. Figure 12-1 shows how IIS communicates with Tomcat using the `isapi_redirect.dll` ISAPI plug-in. There can be one or more instances of Tomcat to serve the client requests.

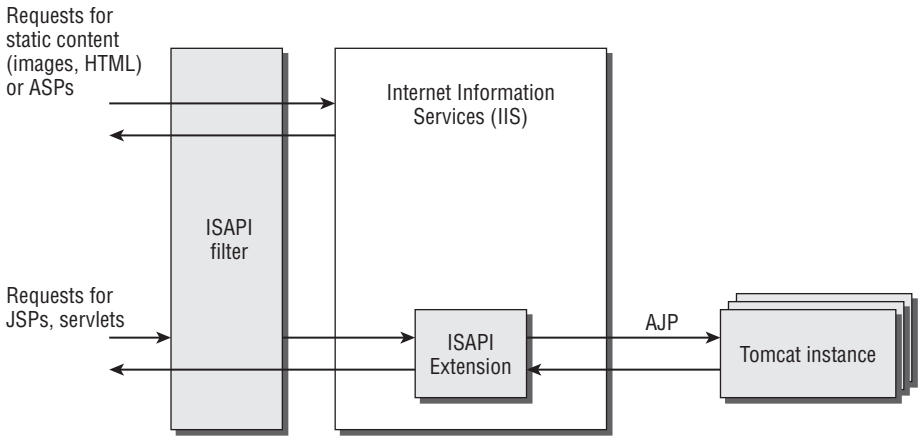

Figure 12-1: The ISAPI plug-in enabling IIS to work as a Web front end to Tomcat

Connecting Tomcat with IIS

Configuring Tomcat and IIS to work together involves a number of steps:

1. Verify your Tomcat and IIS installations.
2. Make Tomcat configuration changes. This includes:
   - ❑ Configure the JK Connector in Tomcat's `server.xml` file.
3. Install and configure the ISAPI plug-in:
   a. Install the ISAPI plug-in.
   b. Configure Tomcat instances or "workers" in the `workers.properties` file.
   c. Configure the request forwarding rules in the `uriworkermap.properties` file.
   d. Optionally, configure URL rewrite rules in a `rewrite.properties` file.
   e. Update the Windows Registry for the ISAPI plug-in.

4. Make IIS configuration changes, such as:
   a. Switch IIS to IIS 5 isolation mode (IIS 6 only).
   b. Create a virtual directory under IIS.
   c. Add the ISAPI plug-in as an IIS filter.
   d. Authorize the ISAPI plug-in as a Web application extension (IIS 6 only).
5. Test the setup (i.e., ensure that IIS redirects requests for JSPs and servlets to Tomcat).

Verifying Tomcat and IIS Installations

Before starting the actual configuration, you should test the installations of Tomcat and IIS.

First confirm that Tomcat is properly configured by performing a quick test: Start Tomcat if it is not already running, and browse to the Tomcat home page (assuming it is running on port 8080):

```
http://localhost:8080/
```

This should display the default Tomcat home page. Chapter 3 covers all you need to know about installing and setting up Tomcat.

Next, start IIS by selecting the following in the Windows menus: Start➪Control Panel➪Administrative Tools➪Internet Information Services. Double-clicking this opens the IIS administration console. In the left pane of the console, expand the top-level server node. Right-click the default Web site and start the server.

If the IIS manager does not show up in Administrative Tools, it is probably not installed. To install IIS, go to your Windows Control Panel, and select Add or Remove Programs. Next, select the Add/Remove Windows Components tab, and make sure the selection for Internet Information Services is checked, as shown in Figure 12-2.

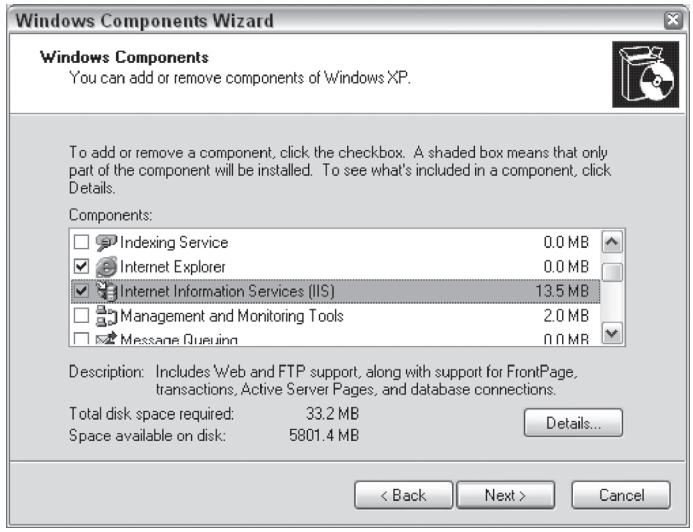
Windows Components Wizard
Windows Components
You can add or remove components of Windows XP.
To add or remove a component, click the checkbox. A shaded box means that only part of the component will be installed. To see what's included in a component, click Details.
Components:
Indexing Service 0.0 MB
Internet Explorer 0.0 MB
Internet Information Services (IIS) 13.5 MB
Management and Monitoring Tools 2.0 MB
Message Queuing 0.0 MB
Description: Includes Web and FTP support, along with support for FrontPage, transactions, Active Server Pages, and database connections.
Total disk space required: 33.2 MB
Space available on disk: 5801.4 MB
Details...
< Back
Next >
Cancel

Figure 12-2: Install IIS.

If you enable IIS, Windows installs the appropriate version of IIS for your OS version. As mentioned earlier, Windows does not give you a choice on which version to install; the version supported for Windows XP Professional and Windows 2000 Server is IIS 5; and IIS 6 is the supported version for Windows Server 2003.

If you are on IIS 6, you need to switch it to the IIS 5 Isolation mode, as explained later in the chapter.

Configuring the JK Connector

Tomcat's `server.xml` file (`<CATALINA_HOME>\conf\server.xml`) contains the configuration entry for the AJP Connector, as shown here. Uncomment the following line if it has been commented out:

```
<!-- Define an AJP 1.3 Connector on port 8009 -->
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

These lines configure an AJP Connector that will use AJP version 1.3 (`protocol` attribute) and will listen on port 8009 (`port` attribute). This listening port is configurable. The other configurable parameters of this Connector are as follows:

- `enableLookups`: If set to `true`, calls to `request.getRemoteHost()` perform DNS lookup to return the actual host name of the remote client. Setting this to `false` skips the DNS lookup and returns the IP address as a string (thereby improving performance). By default, DNS lookups are disabled.
- `redirectPort`: If this Connector is supporting non-SSL requests, and a request is received for which a matching `<security-constraint>` requires SSL transport, Tomcat automatically redirects the request to the port number specified here.
- `scheme`: Set this attribute to the name of the protocol you wish to have returned by calls to `request.getScheme()`. For example, you would set this attribute to `https` for an SSL Connector. The default value is `http`.
- `secure`: If set to `true`, calls to `request.isSecure()` return `true` where the Connector is SSL enabled. The default value is `false`.
- `debug`: This specifies the debugging detail level of log messages generated by this component, with higher numbers creating more detailed output. If not specified, this attribute is set to zero (`0`).
- `protocol`: This attribute value must be `AJP/1.3` to use the JK handler.

Installing the ISAPI Plug-in

The binaries of the ISAPI plug-in can be downloaded from the Tomcat download Web site (`apache.org/dist/tomcat/tomcat-connectors/jk/binaries/win32/`). Under this location, there might be multiple versions of the JK Connector. Select the stable version as indicated on the Web site; the stable JK version at the time of this writing was JK 1.2.21. Under the JK Connector directory, you will find the ISAPI plug-in DLL: `isapi_redirect.dll`.

You can save this DLL anywhere on your file system, and in this example we copy it under `C:\Inetpub\ISAPI\bin`. This directory structure does not exist, and you would need to create the `C:\Inetpub\ISAPI` and `C:\Inetpub\ISAPI\bin` directories.

Most Tomcat documentation online recommends that you copy this plug-in under the Tomcat installation directory, typically under `<CATALINA_HOME>\bin\win32\i386`. This is due to historical reasons, as earlier Tomcat versions shipped with the plug-in in that location. This path does not need to be under the Tomcat install directory: Tomcat need not even be running on the same machine as IIS.

Whatever the location of the plug-in, make a note of this path because you must specify this while creating the "virtual directory" under IIS. Creating a virtual directory is discussed later in this chapter.

Configuring Tomcat Workers

As explained in the previous chapter, a *worker* is an instance of Tomcat that serves up servlets or JSPs on behalf of a Web server, such as IIS or Apache. These workers are configured in a properties file, typically named `workers.properties`, and in our example, stored under `C:\Inetpub\ISAPI\conf`. This directory structure does not exist, and you need to create the `C:\Inetpub\ISAPI\conf` directory.

Again, the Tomcat documentation mentions that the location of the `worker.properties` is `<CATALINA_HOME>\conf`. Because this file is not used by Tomcat, but instead by the ISAPI plug-in — and hence IIS — it is more logical to store it outside the Tomcat install directory. Besides, as mentioned earlier, Tomcat need not even be running on the same server machine as IIS, and if this is the case, then these files need to be on a suitable location on the machine that IIS (and the ISAPI plug-in) is installed on.

Make a note of this file name and location; you need it while configuring the ISAPI plug-in.

Now edit this `workers.properties` file, and add the following lines:

```
worker.list = my-tomcat-worker
worker.my-tomcat-worker.type = ajp13
worker.my-tomcat-worker.host = localhost
worker.my-tomcat-worker.port = 8009
```

In this file, the `worker.list` directive specifies that there is one Tomcat instance, a worker named `my-tomcat-worker`. This is an AJP13 worker, as the `worker.<worker name>.type` directive specifies. This means that it uses Apache JServ Protocol (AJP) version 1.3 for the IIS and Tomcat communication. Chapter 4 explains more about the AJP protocol.

Next, the `worker.<worker name>.host` and `worker.<worker name>.port` directives specify the host and port for the Tomcat instance. The host must correspond to the host name or IP address of the machine that Tomcat is running on. In this example, Tomcat is running on the same server as IIS, so the host parameter is set to `localhost`. The port number corresponds to that configured as the AJP Connector port in Tomcat (see the section "Configuring the JK Connector" earlier in the chapter).

As you can see from this configuration file, the directives in a worker properties file are in the format:

```
worker.<directive>=<value>
```

or

```
worker.<worker name>.<directive>=<value>
```

There can also be more than one worker configured in the properties file, as shown next:

```
worker.list = my-tomcat-worker1, my-tomcat-worker2
worker.my-tomcat-worker1.type = ajp13
worker.my-tomcat-worker1.host = host1
worker.my-tomcat-worker1.port = 8009
worker.my-tomcat-worker2.type = ajp13
worker.my-tomcat-worker2.host = host2
worker.my-tomcat-worker2.port = 8009
```

Why would you want multiple Tomcat workers? For a lot of different reasons: Multiple workers allow for load balancing of requests between different Tomcat instances, or for isolating different Tomcat environments from each other, or for "virtual hosting" environments, where Web sites run on the same physical hardware — usually for cost reasons — but appear as different Web sites to end users. See Chapter 15 for a detailed description of such a virtual hosting situation, and Chapters 11 and 17, for load balanced, clustered configurations.

The following table lists Tomcat worker directives relevant to the JK connector.

Worker Directive	Description
`worker.list`	A list of Tomcat worker names. Multiple workers can be specified by a comma-separated list, or by having the `worker.list` directive repeated.
`worker.maintain`	Specifies the time in seconds for the worker connection pool to be maintained. This defaults to 60 seconds.
`worker.<worker_name>.type`	The type of worker. The type required for connecting Tomcat and IIS is `ajp`. Other types include `lb` (for load balancing workers), `jni`, and `status` (for load balancer management).
`worker.<worker_name>.host`	Host name or IP address of server machine that the Tomcat instance runs on.
`worker.<worker_name>.port`	Port number configured in the AJP Connector in the Tomcat instance. This defaults to 8009.
`worker.<worker_name>.socket_timeout`	Timeout value in seconds for connections between IIS and Tomcat. A value of 0 will wait indefinitely.
`worker.<worker_name>.socket_keepalive`	Send `KEEP_ALIVE` messages between IIS and Tomcat or not. This parameter defaults to `False` (i.e., don't send `KEEP_ALIVE` messages). Keep alive is typically required when the connection between IIS and Tomcat is across a firewall, and hence there is a risk of inactive connections being dropped.
`worker.<worker_name>.retries`	Number of retries the work does if a connection request between IIS and Tomcat failed. This defaults to 2.
`worker.<worker_name>.connection_pool_size`	The connections made between IIS and Tomcat are kept in a connection pool. This parameter allows for the maximum connection pool size to be configured.

Worker Directive	Description
`worker.<worker_name> .connection_pool_minsize`	The connections made between IIS and Tomcat are kept in a connection pool. This parameter allows for the minimum size of the connection pool to be configured. This parameter defaults to `(connection_pool_size+1)/2`.
`worker.<worker_name> .connection_pool_timeout`	This property is used when the `connection_pool_size` is used. It specifies how many seconds an inactive socket connection is kept in the connection pool before closing it. This property helps reduce the number of threads on the Tomcat Web server. A value of 0 disables the connection timeout.

Configuring the Request Forwarding Rules

The previous section explained how Tomcat's workers were configured. How does the Web server — IIS or Apache — know which requests have to be forwarded to the Tomcat worker(s)? This is done via another property file, called the `uriworkermap` file.

To test your install of the ISAPI redirector, configure Tomcat to serve up content from the example Web application that is shipped with Tomcat. To do this, create a file called `uriworkermap.properties` with the contents that follow, and store it under `C:\Inetpub\ISAPI\conf`. Again, as in the case of `workers.properties`, this file name and location are a convention, and not a requirement. However, make a note of this file name and location as you need this information while configuring the ISAPI plug-in.

```
/examples/*= my-tomcat-worker
```

This line tells the ISAPI plug-in to forward all requests for the `examples` Web application to the Tomcat worker named `my-tomcat-worker` for processing. This was the name for the worker defined earlier in the `workers.properties` file.

As you can see, the general patterns of directives in the `uriworkermap` file are in the format:

```
<pattern>=<worker name>
```

The patterns allowed can be fairly complex; they can specify URL pattern, wildcards, exclusions, and rules to support virtual hosts. Some examples of other patterns that can be used are listed in the following table.

Worker Directive	Description
`*.jsp=<worker name>`	Forwards all requests for a URI matching the specified suffix (i.e., all JSP files) to the specified worker.
`/examples/=<worker name>`	Forwards all requests to the URI `/examples` to the specified worker.

Table continued on following page

Worker Directive	Description
`/examples/*=<worker name>`	Forwards all requests for everything *under* the URI `/examples` to the specified worker. The * wildcard matches any number of characters in the URI, and the ? wildcard matches exactly one character.
`/examples|/*=<worker name>`	Forwards all requests for the URI `/examples` *and* everything under it to the specified worker.
`!/examples/static|/*=<worker name>`	Exclusion rule: Don't send any requests matching the specified URI (everything under `/examples/static`) to the specified worker.
`!*.html=<worker name>`	Exclusion rule: Don't send any requests matching the specified URI (i.e., any HTML file) to the specified worker.
`/www.example.com/examples/*=<worker name>`	Rule for supporting virtual hosts (IIS only): Sends all requests for `examples` Web application on the `www.example.com` virtual host to the specified worker.

Optionally Configure URL Rewrite Rules

The ISAPI redirector also supports simple URL rewriting. This is done by writing the URL rewrite rules in yet another properties file.

To do this, create a file called `rewrite.properties` with the contents shown below, and store it under `C:\Inetpub\ISAPI\conf`. Again, as in the case of `workers.properties`, this file name and location are a convention, and not a requirement. However, make a note of this file name and location as you need it while configuring the ISAPI plug-in.

```
/servlets-examples/=/examples/servlets/
```

The previous example redirects all requests with the URI pattern `/servlets-examples` to the `/examples/servlets` URI.

As you can see, the pattern is of the form:

```
<Requested URI>=<Replacement URI>
```

Updating the Windows Registry for the ISAPI Plug-in

Windows uses the Registry system to store configuration information about any system component. IIS also uses the Windows Registry as a reference for getting information about its extensions. The ISAPI plug-in settings also need to be stored in the Windows Registry. This can be done either manually or by running a Registry script.

> *It is highly recommended that you make a backup of the current Registry before proceeding further. This will enable you to restore the original configuration if anything goes wrong.*

One simple way to make a backup of your Registry is to use the File➪Export option in the Registry Editor (`regedit`). To restore the Registry, simply use the File➪Import option to get back to the original setting.

Editing the Registry Manually

To update the Windows Registry manually, select Start➪Run from the Windows menu. Type the command **regedit** in the command box and click OK, as shown in Figure 12-3.

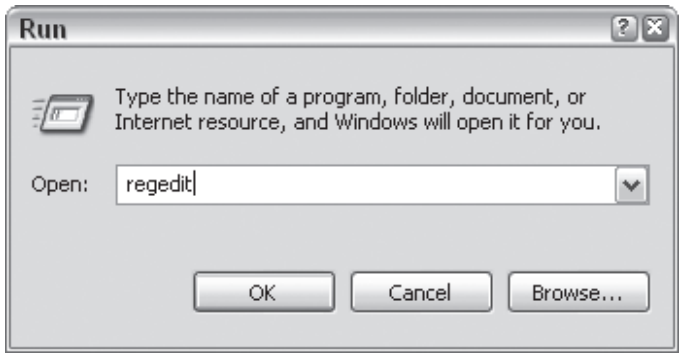

Figure 12-3: Invoking regedit

The execution of the `regedit` command opens the Windows Registry. The Registry is a hierarchical collection of keys. The left pane shows the Registry entries as a tree. The right pane shows associated subkeys for a selected key in the left panel. A typical Windows Registry is shown in Figure 12-4.

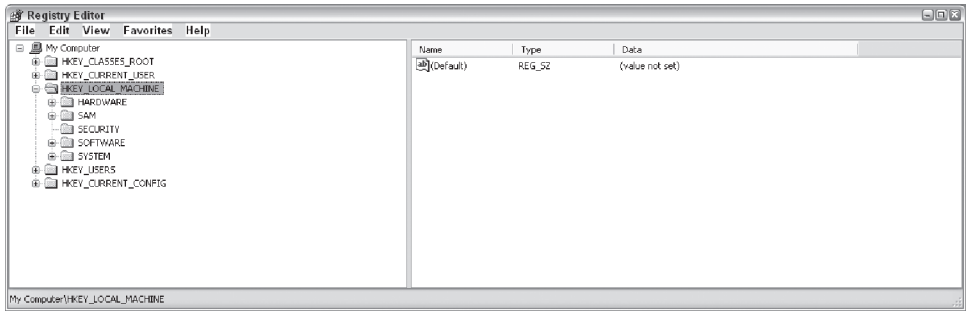

Figure 12-4: The Windows Registry Editor window

To add the required new keys in the Windows registry, first locate the `HKEY_LOCAL_MACHINE -> Software` branch in the left pane.

Now, create a new key called `Apache Software Foundation` under Software by right-clicking and selecting New➪Key. Under this, create another key named **Jakarta ISAPI Redirector.** Finally, create another key named **1.0** under `Jakarta ISAPI Redirector`. You might already have a key for `Apache Software Foundation` and `Jakarta ISAPI Redirector` in your Registry if you installed Tomcat using the installer executable, or configured the ISAPI plug-in previously.

After this is done, you are ready to add the configuration parameters for the ISAPI plug-in. Right-click on the branch 1.0 and set the following parameters using the New➪String Value option:

- `extension_uri`: Enter the value `/tomcat/isapi_redirect.dll`. Here, *`tomcat`* refers to the name of the IIS virtual directory that you create later. You can use any name that you prefer.

- `log_file`: Enter the path to a log file where the ISAPI redirector will send log messages, for example, `C:\Inetpub\ISAPI\logs\iis_redirect.log`. The `C:\Inetpub\ISAPI\logs` directory does not exist, and needs to be created. This path, and all other paths specified in the ISAPI Registry entry must be absolute paths.
- `log_level`: The desired level for the log message — set this to `debug` for now. The other possible values are `debug`, `info`, `warn`, `error`, and `trace`. Once the configuration is working properly, you can raise this level to `warn` or `error`.
- `worker_file`: This is the path to the Tomcat workers configuration file that was mentioned earlier — for example, `C:\Inetpub\ISAPI\conf\workers.properties`.
- `worker_mount_file`: This is the path to Tomcat's request forwarding configuration file that was mentioned earlier — for example, `C:\Inetpub\ISAPI\conf\uriworkermap.properties`.
- `rewrite_rule_file`: This is the path to the optional rewrite rule file that was mentioned earlier — for example, `C:\Inetpub\ISAPI\conf\rewrite.properties`.

After you have made these setting changes, your Registry Editor should look like that in Figure 12-5.

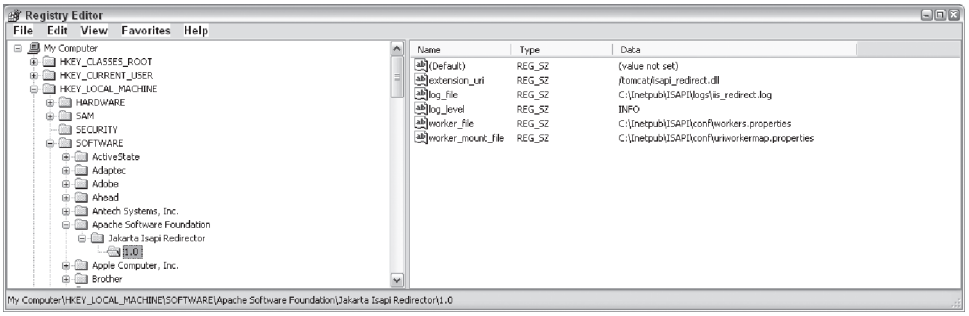

Figure 12-5: The Registry settings for the ISAPI plug-in

Of these settings, the mandatory ones are `extension_uri`, `worker_file`, and `worker_mount_file`. There are other optional settings too, as listed here:

- `shm_size`: This is the size allocated for the shared memory used by the Tomcat workers. In most configurations you would not need to set this value. If you have a large number of workers, say more than 64, you should set this to `<number of workers> * 400`. While adding the `shm_size` parameter, use the New➪DWORD Value and not the String Value option.
- `worker_mount_reload`: This specifies the time in seconds after which the `worker_mount_file` will be reloaded. While adding this parameter, use the New➪DWORD Value option. Again, this is something that is rarely configured.
- `strip_session`: Set this value to `T`, `t`, or `1` for stripping the session suffixes of the form `;jsessionid=...` from URLs. Set this parameter value to `F`, `f`, or `0` for `false`. This is a string valued parameter — i.e., use New➪String Value to add it.

A less error-prone mechanism to update the Registry is to update it using a script, as shown next.

Editing the Registry via a Script

The Windows Registry can also be updated at one pass by running a script. To update the Registry using a script, create the following script and save it as `isapi.reg`. You may need to change some parameters in the script, such as the location of your log file, `workers.properties`, and `uriworkermap.properties`.

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SOFTWARE\Apache Software Foundation\Jakarta Isapi
Redirector\1.0]
"extension_uri"="/tomcat/isapi_redirect.dll"
"log_file"="C:\\Inetpub\\ISAPI\\logs\\iis_redirect.log"
"log_level"="INFO"
"worker_file"="C:\\Inetpub\\ISAPI\\conf\\workers.properties"
"worker_mount_file"="C:\\Inetpub\\ISAPI\\conf\\uriworkermap.properties"
```

The script file should be edited in a plain text editor such as Notepad. Note that the double quotation marks used in the script file should be the double quotation marks used in a plain ASCII text file. The quotation marks used in a word editor such as Microsoft Word will introduce problems in the execution of the script.

Modify the contents of the script as per your local settings and save it on the disk. Run the script for updating the Windows Registry by double-clicking the script file. This will pop up a message box, as shown in Figure 12-6.

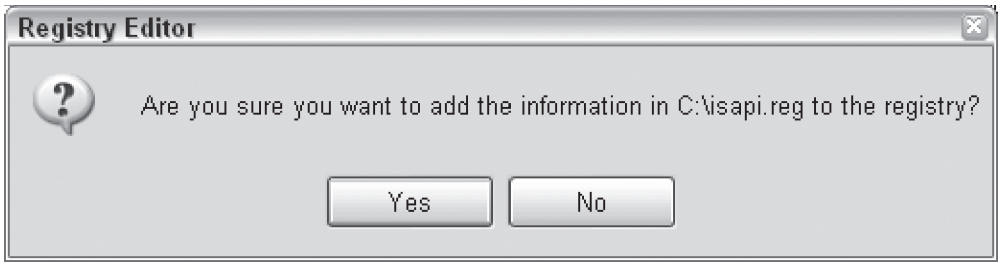

Figure 12-6: The Registry settings for the ISAPI plug-in

Select Yes and proceed. On execution, the script creates the entries in the Windows Registry. You can check to see if the entry was created correctly by running regedit and looking at the entries under `HKEY_LOCAL_MACHINE\SOFTWARE\Apache Software Foundation\Jakarta Isapi Redirector\1.0`. You should see Registry entries similar to those in Figure 12-5.

IIS 5 Isolation Mode (IIS 6 Only)

For Tomcat support in IIS 6 (i.e., if you are on Windows Server 2003), you would need to switch IIS to the *IIS 5 isolation mode*. IIS 6 runs a server in one of two distinct request processing models, also called application isolation modes. These include the *Worker process isolation mode*, and the IIS 5 isolation mode. In the Worker process isolation mode, IIS behaves like a regular IIS 6 server and you can use the new features provided by IIS 6 for security and reliability, including the ability to specify configuration settings for a group of applications, and run applications in an isolated environment. The IIS 5 Isolation mode is used for backward compatibility for applications developed on IIS 5, and for Tomcat support.

To switch isolation modes, perform the following steps:

1. Right-click and select Properties on the Web Sites node in the IIS Manager.
2. Select the Service tab in the Properties page.
3. Check the check box for running WWW service in IIS 5 isolation mode, as shown in Figure 12-7.

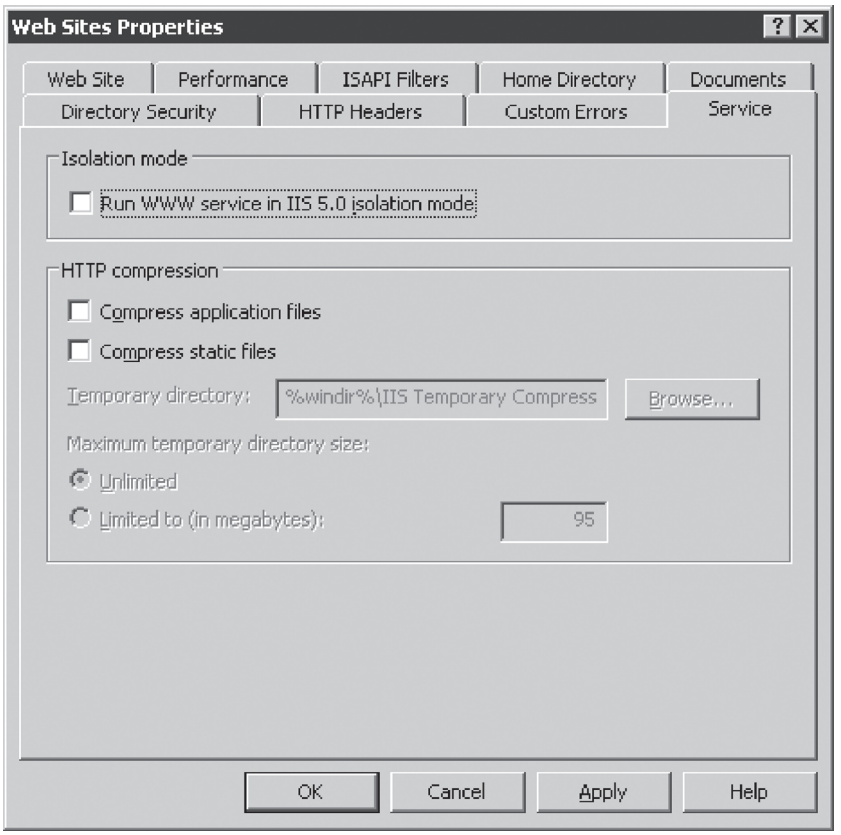

Figure 12-7: Switching to the IIS 5 Isolation mode (IIS 6 only)

4. Restart IIS for the changes to take effect.

Creating a Virtual Directory Under IIS

IIS needs to locate the ISAPI plug-in as a server extension. Adding a virtual directory under IIS and loading the ISAPI plug-in addresses this requirement.

The first part of this process is to add a virtual directory. The IIS manager provides a wizard for this task. In Windows Control Panel select Administrative Tools and then the Internet Services Manager. The left panel displays the components of the server. Expand the nodes and select the Default Web Site node.

Add a virtual directory by right-clicking the Default Web Site node and select New➪Virtual Directory, as shown in Figure 12-8.

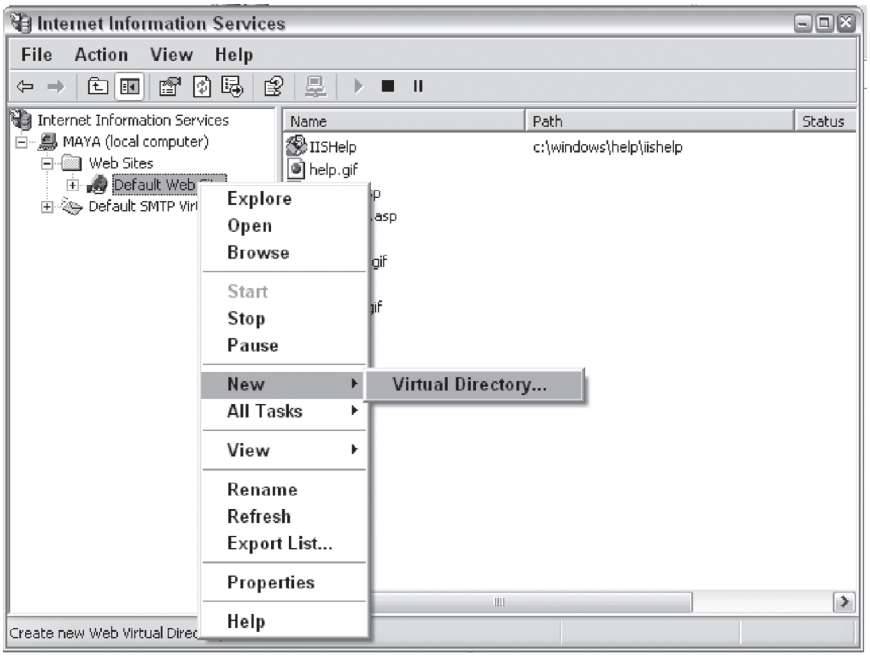

Figure 12-8: Starting the Virtual Directory Creation Wizard

This will start the wizard, as shown in Figure 12-9.

Virtual Directory Creation Wizard
Welcome to the Virtual Directory Creation Wizard
This wizard will help you create a new Virtual Directory on this Web site.
Click Next to continue.
< Back
Next >
Cancel

Figure 12-9: The Virtual Directory Creation Wizard

After you click the Next button, the next screen requests an alias name for the virtual directory (see Figure 12-10). Enter **tomcat** as the alias name. This is the same name that was specified earlier in the Windows Registry in the `extension_uri` parameter prefix (i.e., as `/tomcat/isapi_redirect.dll`).

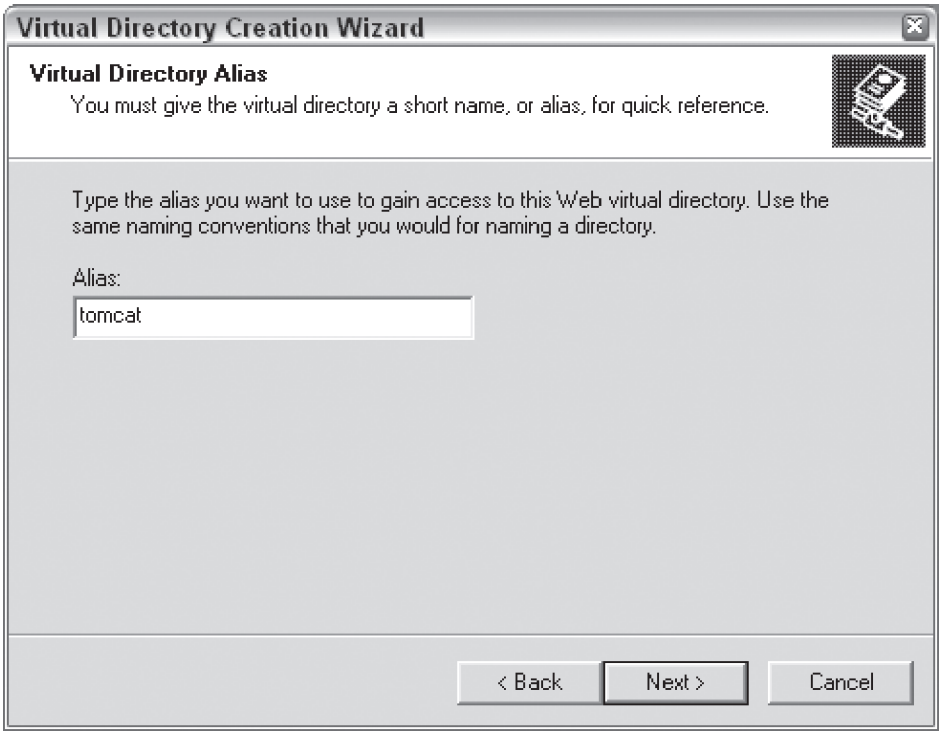
Virtual Directory Creation Wizard
Virtual Directory Alias
You must give the virtual directory a short name, or alias, for quick reference.
Type the alias you want to use to gain access to this Web virtual directory. Use the same naming conventions that you would for naming a directory.
Alias:
tomcat
< Back
Next >
Cancel

Figure 12-10: Specifying the virtual directory alias

After you click Next, the wizard asks for the actual location for the virtual directory. Click the Browse button to specify the directory in which the ISAPI plug-in is located. For the configuration discussed in this chapter, it is `C:\Inetpub\ISAPI\bin`. Again, confirm that this was the location to which the ISAPI plug-in was copied (see the section "Installing the ISAPI Plug-in" earlier in this chapter).

In the next screen, select the permissions that you want to provide for accessing the directory. Select options Read, Run Scripts, Execute (such as ISAPI applications or CGI), and Browse, as shown in Figure 12-11.

> *The* `Browse` *option enables directory browsing for the virtual directory. The* `Browse` *option can be disabled later, after the setup is tested.*

Complete the final step of the wizard. On completion, the virtual directory named tomcat will be displayed as a sub-branch.

After creating the virtual directory, check all the values specified by right-clicking the virtual directory and selecting the Properties option. A screen with all the details is displayed, as shown in Figure 12-12.

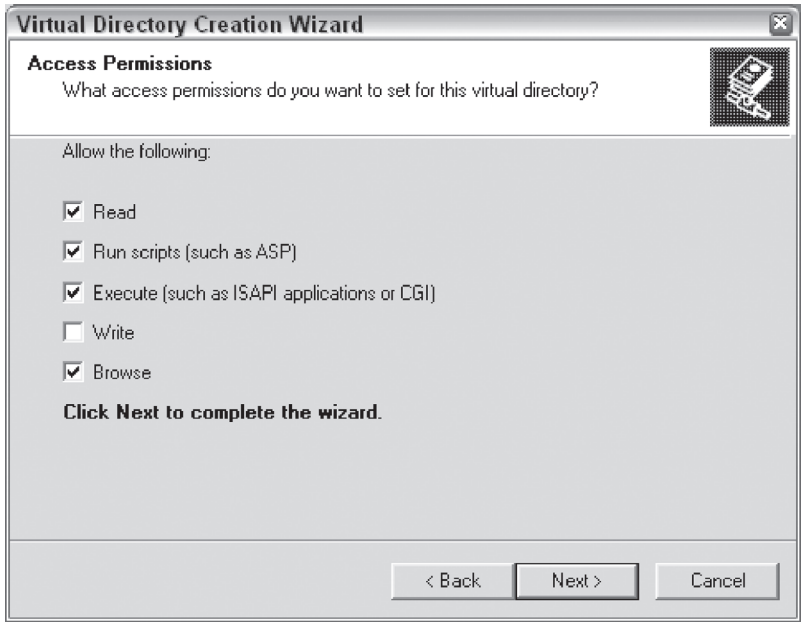

Figure 12-11: Set the virtual directory access permissions

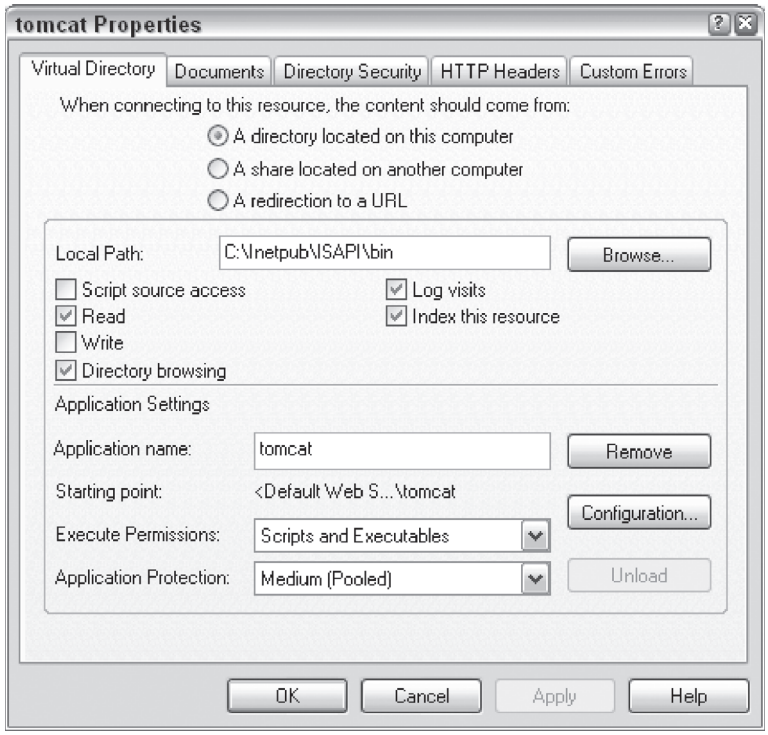

Figure 12-12: Checking the virtual directory settings

Adding the ISAPI Plug-in as an IIS Filter

The next part of the configuration is to load the ISAPI plug-in as an extension to the IIS main Web service. As before, start the Internet Information Services Manager from Administrative Tools.

Now, right-click on the Default Web site and select Properties. This opens the Default Web Site properties dialog box. Click the ISAPI Filters tab and click the Add button to add a new ISAPI filter. Another dialog box appears, requesting the file name and the location of the executable of the ISAPI plug-in to be added, as shown in Figure 12-13. Enter a name for the ISAPI plug-in, such as `isapi_redirect`, and provide the location of the ISAPI plug-in, i.e., `C:\Inetpub\ISAPI\bin\isapi_redirect.dll`. The name of the plug-in can be anything, and is not tied to any other configuration file.

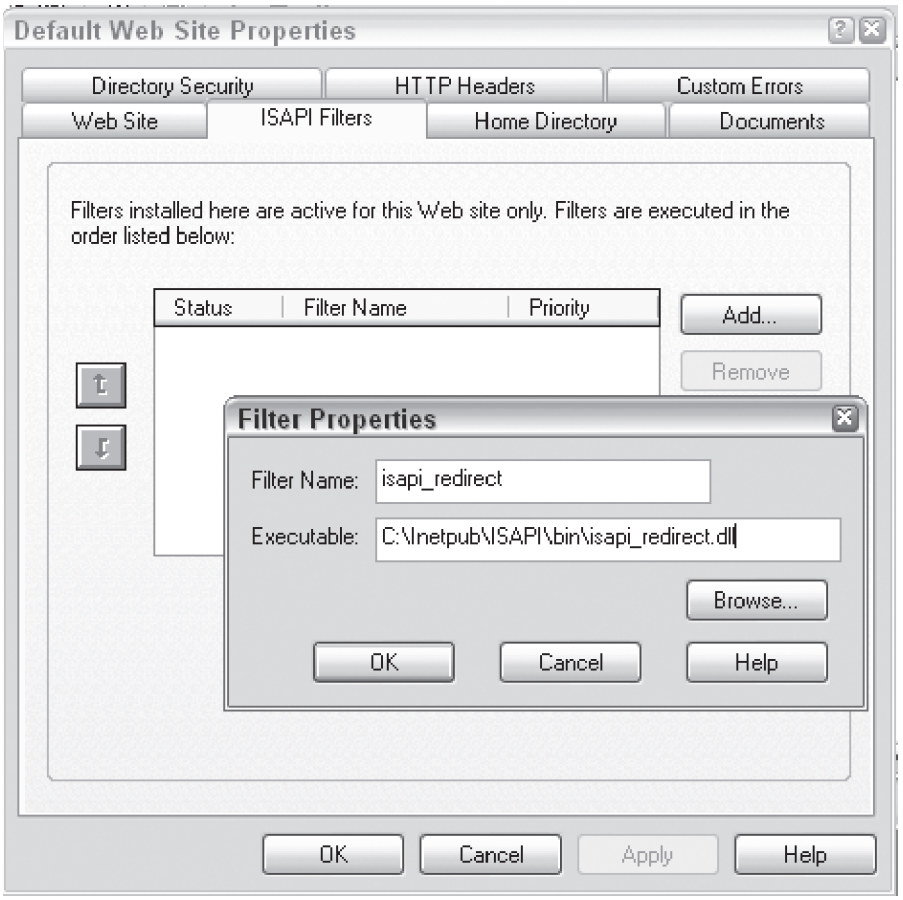

Figure 12-13: Adding the ISAPI plug-in as a filter to IIS

Click OK to add the filter. At this stage, the filter is not yet usable by IIS. This unavailability is indicated by the missing upward-pointing green status arrow for this filter in the ISAPI filter list, as shown in Figure 12-14.

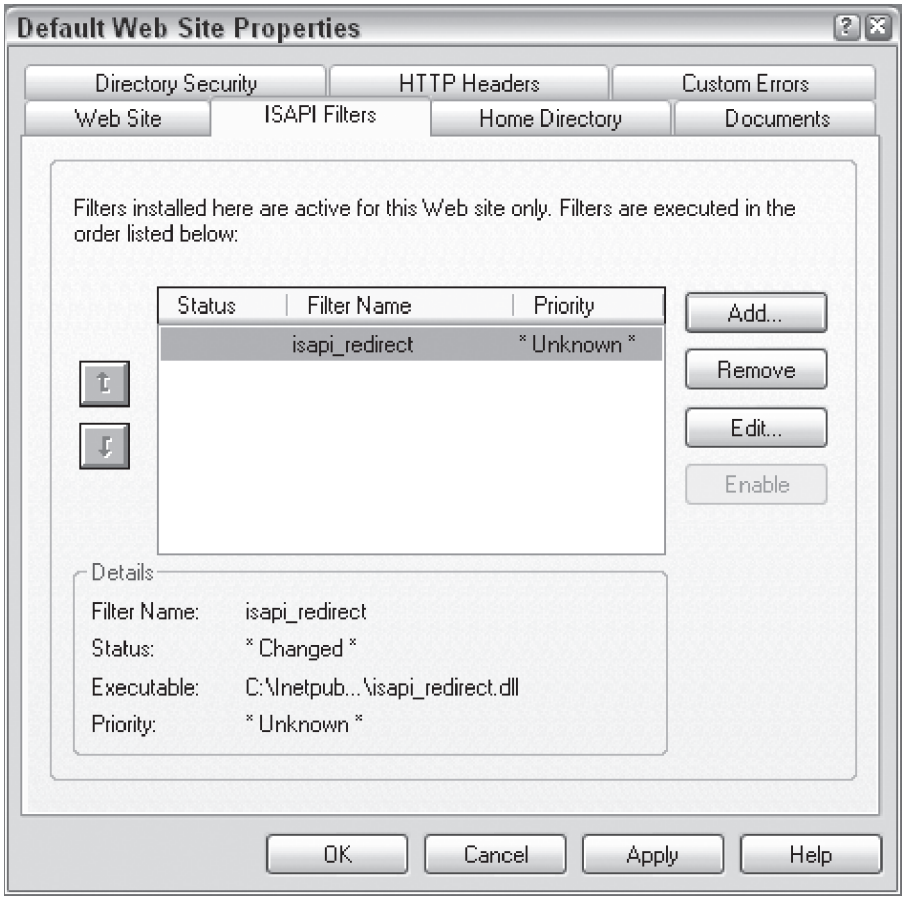

Figure 12-14: Adding the ISAPI plug-in as a filter to IIS

Click Apply and then stop and start IIS by right-clicking the Default Web Site node. After restarting IIS, check the list of ISAPI filters again. This time the ISAPI plug-in is displayed with an upward-pointing green status arrow, as shown in Figure 12-15.

The ISAPI plug-in is now loaded and IIS can use it as a filter for all the incoming requests.

In case of failure, you will see a downward-pointing red arrow. Check the Windows Registry entries and verify the `isapi_redirect.dll` path that was specified while adding the ISAPI filter.

> *The ISAPI plug-in can also be configured as a filter at the Main Server instead of the Default Web Site. If the setup is not working after setting up the ISAPI filter for the Default Web Site, try setting up the ISAPI filter at the Main Server.*

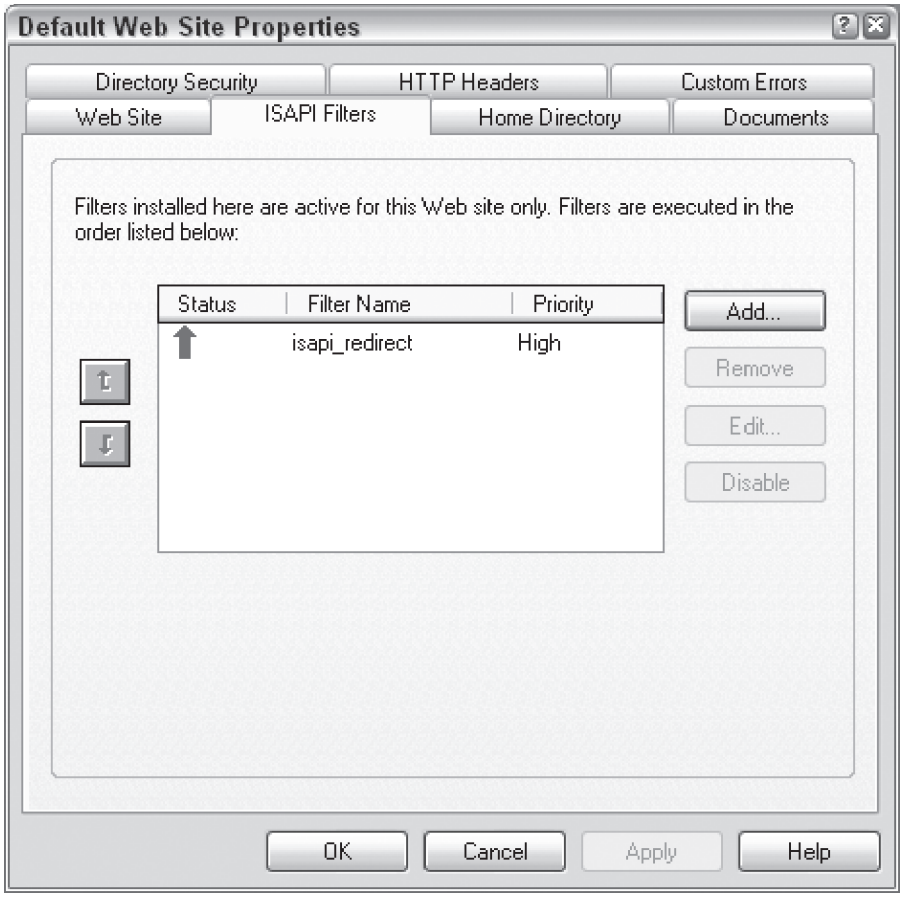

Figure 12-15: Successful installation of the ISAPI plug-in as a filter

Authorizing the ISAPI Plug-in as a Web Application Extension (IIS 6 Only)

Another IIS 6–specific step is to authorize the ISAPI DLL as a Web application. The steps involved in this are as follows:

1. Start the IIS console as before (Start➪Control Panel➪Administrative Tools➪Internet Information Services), and click on Web Service Extensions in the left-hand menu tree.
2. Right-click and select Add a new Web Service Extension.
3. Enter a name for the extension, and then click Add.
4. Select the ISAPI DLL (`C:\Inetpub\ISAPI\bin\isapi_redirect.dll` in this example setup).
5. Check the Set extension status to Allowed check box.
6. Stop and start IIS by right clicking on the Default Web Site node.

Testing the Final Setup

After successfully completing the previous steps, you are now ready to test connectivity between Tomcat and IIS.

Executing any of the sample JSP or servlets under the `examples` context tests the final setup. Point your browser to the following location:

```
http://localhost/examples/jsp/
```

This should show a list of JSP samples, as shown in Figure 12-16. Execute any one of them, and they should run successfully.

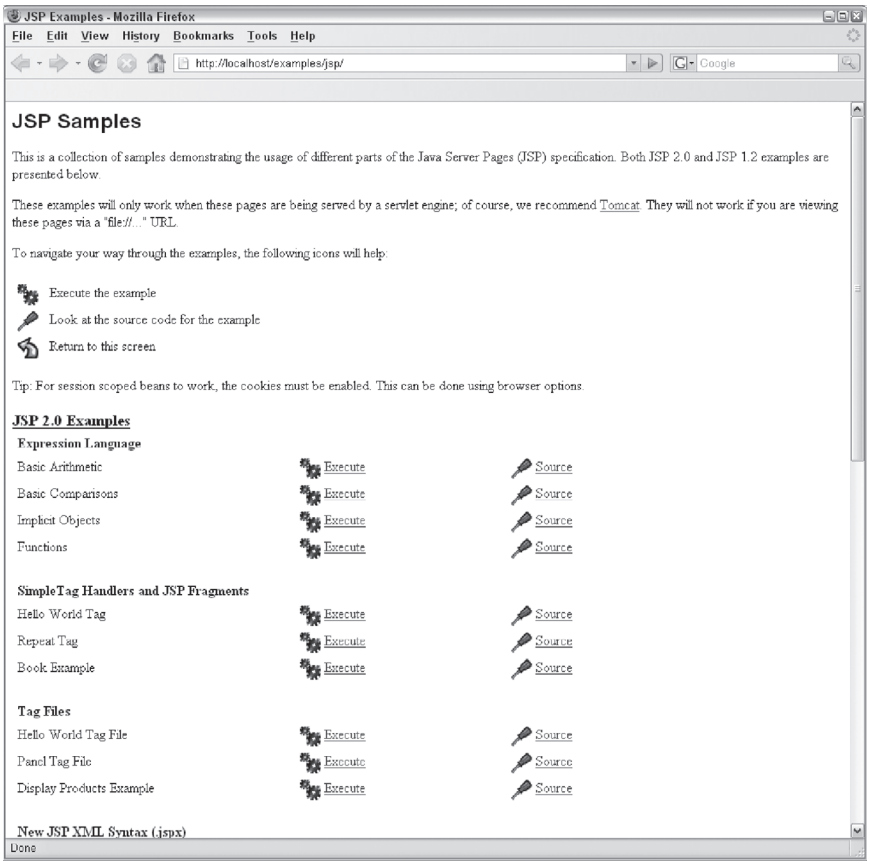
JSP Examples - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/examples/jsp/
JSP Samples
This is a collection of samples demonstrating the usage of different parts of the Java Server Pages (JSP) specification. Both JSP 2.0 and JSP 1.2 examples are presented below.
These examples will only work when these pages are being served by a servlet engine; of course, we recommend Tomcat. They will not work if you are viewing these pages via a "file://..." URL.
To navigate your way through the examples, the following icons will help:
Execute the example
Look at the source code for the example
Return to this screen
Tip: For session scoped beans to work, the cookies must be enabled. This can be done using browser options.
JSP 2.0 Examples
Expression Language
Basic Arithmetic Execute Source
Basic Comparisons Execute Source
Implicit Objects Execute Source
Functions Execute Source
SimpleTag Handlers and JSP Fragments
Hello World Tag Execute Source
Repeat Tag Execute Source
Book Example Execute Source
Tag Files
Hello World Tag File Execute Source
Panel Tag File Execute Source
Display Products Example Execute Source
New JSP XML Syntax (jspx)
Done

Figure 12-16: Testing the setup with a successful JSP execution

Troubleshooting Tips

The procedure described in this chapter for configuring Tomcat with IIS is somewhat involved and tricky, especially since the error messages leave a lot to be desired. This section covers some common pitfalls and provides debugging tips to help resolve configuration-related issues.

If all the configuration steps were followed, but you still experience problems with IIS rendering JSPs, use the following guidelines to troubleshoot the most common problems:

- Check whether the IIS Web server is running. This can be tested by visiting the IIS home page. You should get the default IIS home page at `http://localhost`. If the use of `localhost` as a server name doesn't work, then try using the actual server name or IP address.
- Similarly, check if Tomcat started properly. Common errors here are around port number clashes — for instance, Tomcat and the Windows Media Services for IIS on Windows 2003 Server use the same default port number (8080). You can have Tomcat listen on a port other than 8080 by editing the `<CATALINA_HOME>\conf\server.xml` configuration file.
- Check if Tomcat's AJP Connector is uncommented. You can also check if Tomcat is indeed listening on the AJP Connector port by using the `netstat` command.
- Check the Registry entries for `HKEY_LOCAL_MACHINE\SOFTWARE\Apache Software Foundation\Jakarta Isapi Redirector\1.0`. You should see Registry entries similar to those in Figure 12-5. Confirm that each of the files referred to by `extension_uri`, `worker_file`, `worker_mount_file`, and `log_file` actually exist. This and an improperly installed ISAPI plug-in are by far the most common problems for a non-functioning setup.
- Confirm that the ISAPI plug-in is properly installed. In the left pane of the Internet Services Manager, right-click the Default Web Site node and then select the Properties option. Click the ISAPI Filters tab. In the list of ISAPI filters, check if the status column for the ISAPI plug-in has a green upward-pointing arrow. If you do not see the green arrow, or see a red arrow, then there is a problem with the ISAPI plug-in installation. Check the Windows Registry entries and verify the `isapi_redirect.dll` path that was specified while adding the ISAPI plug-in. The plug-in can be added as a filter for all Web sites, or a specific Web site (such as the Default Web site). If one doesn't work, try the other.
- Verify that the virtual directory (`tomcat`) is defined properly in IIS. If something is wrong with it, IIS will indicate this by flagging it with a red symbol.
- Verify that the name of this virtual directory matches the prefix name specified in the Registry for the `extension_uri` (in the example setup, the name `tomcat` was used).
- Also confirm that you have given Script and Execute permissions to the virtual directory, as shown in Figure 12-2.
- Check the ISAPI filter log file for errors. This file was specified in the `log_file` parameter in the Registry. In this example, it is the `C:\Inetpub\ISAPI\log\iis_redirect.log` file.
- The following sample log shows the error messages when the ISAPI cannot connect to the Tomcat worker named "my-tomcat-worker." This could mean anything from an incorrect worker host/port specified in the worker properties file, Tomcat AJP Connector not configured, or Tomcat not running.

```
[Sun Apr 15 23:08:29 2007] [3848:2192] [info]  jk_ajp_common.c (876): Failed
opening socket to (192.168.1.2:8009) (errno=61)
[Sun Apr 15 23:08:29 2007] [3848:2192] [info]  jk_ajp_common.c (1273): (my-tomcat-
worker) error connecting to the backend server (errno=61)
[Sun Apr 15 23:08:29 2007] [3848:2192] [info]  jk_ajp_common.c (1930): (my-tomcat-
worker) sending request to tomcat failed,  recoverable operation attempt=2
[Sun Apr 15 23:08:29 2007] [3848:2192] [error] jk_ajp_common.c (1942): (my-tomcat-
worker) Connecting to tomcat failed. Tomcat is probably not started or is listening
 on the wrong port
```

- Check the IIS Web server log file for errors. By default, the IIS server log is located at `C:\Windows\system32\LogFiles\W3SVC1\`. The IIS log file location can be changed using the IIS console — right-click on the Web site, select Properties, and then select the Web Site tab. Click the Properties button next to the Enable Logging check box to configure the location of the log file.

Using SSL

Secure Socket Layers (SSL) provides a secure communication channel between the browser and the Web server. Chapter 10 covers how SSL is set up for the HTTP Connector. When IIS is used with Tomcat, you can use SSL at either the IIS end or the Tomcat end. Using SSL with IIS is often preferred, as other Web pages and deployed applications (for example, ASP applications) on the same server can use it, too.

- **Configuring SSL in Tomcat:** This configuration is explained in detail in Chapter 14. In brief, it involves the following steps:
  1. Download and install an SSL implementation.
  2. Create a certificate keystore and add a self-signed certificate to it.
  3. Purchase a certificate from a certificate authority, such as VeriSign or Trustcenter. The self-signed certificate created in the preceding step is used to generate a certificate-signing request.
  4. Configure Tomcat for SSL.
- **Configuring SSL in IIS:** Refer to the documentation provided with your IIS installation for details.

Scalable Architectures with IIS and Tomcat

So far, we have talked about setting up a configuration where IIS and Tomcat are both running on the same physical server. This is useful for smaller implementations, or development setups. However this is not appropriate for production environments where there are scalability considerations.

There are a number of architectural variants to this configuration that provide higher scalability:

- Partition your deployment into multiple tiers with the Web tier (static content served by IIS) and application tier (dynamic content served by Tomcat) that reside on a separate servers. This allows you to "scale out" each tier independently. Figure 12-17 illustrates this architecture.
- Multiple IIS Web servers can be deployed on a Web server farm, which are then load-balanced using a load-balancing switch.
- Multiple application servers can be deployed, too, each running one or more Tomcat instances.
- The multiple Tomcat workers themselves can be load balanced.

All these configurations are possible with the AJP protocol and the concepts remain the same whether you are using Apache or IIS as the Web server. Chapters 11 and 17 cover configuration details for these architectures. Even though these chapters deal with Apache as the Web server front end to Tomcat, the concepts and configurations translate very well to IIS.

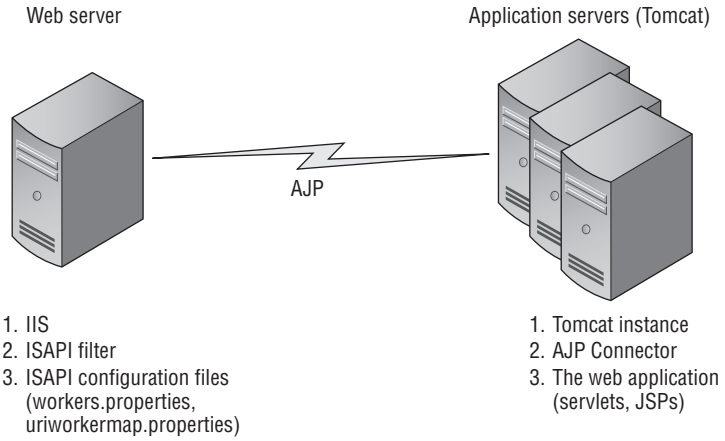

Figure 12-17: Distributed Architecture with IIS and Tomcat on different servers

Finally, notice that once you begin partitioning your architecture into Web server and application server configurations, you have paved the way for a heterogeneous system, so even though you use IIS for the Web server, your application server can be Unix/Linux based.

Distributing Web and Application Server Deployments

Let's start with a simple example and build on it. Consider Figure 12-17, where the IIS is on one server and a Tomcat worker is running on a different server. How would you configure IIS and Tomcat to work with this configuration? The client server design of the AJP13 protocol makes this quite simple to configure.

First, look at the software stack on each sever. The *Web server*, which has IIS running on it, will also have the `isapi_redirect.dll` and the configuration files (`uriworkermap.properties` and `workers.properties`). Note that the JDK and the Tomcat binaries are not required on the Web server. The *Application server* will have the full Tomcat install, along with the JDK. The AJP13 connector must be configured on the application server and a Tomcat worker must be running on a known port on the application server. Also note that if you have multiple application servers with Tomcat workers distributed across these application servers servicing the same Web application, you would want to duplicate the Web application files across all these servers.

Next, look at the configuration changes required. From a conceptual perspective, it is probably instructive to look at a non-distributed configuration where everything is on one box and then examine what needs to be done to split it up into a distributed configuration. Let's look at a specific example, where a single worker called `my-tomcat-worker` is servicing the `examples` application. This is a good example because you already went through the steps required to install this specific configuration on a single box in an earlier section. Now look at the changes that would be required if you were to install this configuration on a Web server and an application server:

1. Install the software on the appropriate servers.
2. Install the software as shown in the previous figure.

3. Modify the `workers.properties` file. The `workers.properties` file will have to be changed so that the host name (or IP address) of the Tomcat worker is the application server, as shown here:

```
worker.myworker.host=myappserver
```

All other entries remain the same. Note that the application server, `myappserver` in this example, must be IP addressable, from the Web server. This example of a distributed configuration of a server running IIS and another server running a single instance of Tomcat is a good demonstration of the client server concepts that form the foundation of the AJP 1.3 protocol.

Multiple Tomcat Workers

As shown earlier in the chapter, supporting multiple Tomcat workers requires the following steps:

1. Configure the AJP Connectors for the workers in Tomcat's `server.xml` file.
2. Add multiple workers to the workers property file as a comma-separated list, as shown:

```
worker.list = my-tomcat-worker1, my-tomcat-worker2
worker.my-tomcat-worker1.type = ajp13
worker.my-tomcat-worker1.host = host1
worker.my-tomcat-worker1.port = 8009
worker.my-tomcat-worker2.type = ajp13
worker.my-tomcat-worker2.host = host2
worker.my-tomcat-worker2.port = 8009
```

The Tomcat workers can be running on separate hosts, as shown in the preceding code, or even on the same host. See Chapter 15 for information on how to run multiple Tomcat instances on the same host.

Load-Balanced AJP Workers

In addition to the AJP worker, a load-balanced worker consisting of multiple `ajp13` workers can be defined. Chapter 11 describes the details for setting up a load-balanced environment. The concepts discussed in Chapter 11 are specified for Apache and Tomcat, but are applicable regardless of the Web server used. More advanced load balancing configurations are covered in Chapter 17.

Summary

This chapter presented details about using IIS as a Web front end to Tomcat.

To conclude this chapter, let's review some of the key points discussed. The main focus of the chapter was configuring IIS to work as a Web front end to Tomcat using the ISAPI. As you saw, this works in the following way:

- IIS is configured with the ISAPI plug-in, which works both as an ISAPI filter as well as a Web application extension.
- The plug-in intercepts all requests to the IIS Web application, and then uses the URI-to-Tomcat-worker mappings defined in the `uriworkermap` file to decide which request to forward

to Tomcat. The requests that are not forwarded to Tomcat are served by the IIS Web server. Typically these would be requests for static resources, such as HTML files, images, and style sheets, as well as dynamic pages developed in Microsoft technologies such as ASP or C#.

- Request for JSPs and servlets can be forwarded by the ISAPI plug-in to Tomcat using its Web application extension features. The specific instance of Tomcat that it forwards to is defined in the `workers` property file.
- The ISAPI extension is responsible for gathering all request parameters as well as handling the response returned by Tomcat. This response is then returned to the user's Web browser.

In addition, the chapter covered how to extend this setup for highly scalable Web deployment architectures, as well as listed troubleshooting tips for common configuration issues.

The next chapter covers JDBC connectivity in Tomcat.

13

JDBC Connectivity

Most Web applications process data, and that data is often stored in a database. The most popular database management systems are based on relational concepts, and are appropriately called *relational database management systems* (or *RDBMSs*).

All popular databases (including Oracle, MySQL, SQL Server, Sybase, Interbase, PostgreSQL, and DB2) are relational databases. Tomcat administrators must be well versed in RDBMSs. In addition, an understanding of the nature of interactions between an RDBMS and Tomcat is required to better anticipate the requirements that may arise.

This chapter addresses the following topics:

- Java Database Connectivity (JDBC), which is Java's database connectivity API
- JDBC version evolution
- JDBC driver types and advantages
- Importance of connection pooling
- Interactions between RDBMS and Tomcat
- JNDI-based JDBC configurations
- Standard configuration for a JDBC data source
- Alternative JDBC configurations that may be required
- Configuring alternative JDBC access

This chapter also covers a variety of situations that may arise when configuring Tomcat to work with relational databases. The examples offered in this chapter feature hands-on configuration. Special emphasis is placed on the recommended or preferred way of interacting with databases.

After reading this chapter, you will be comfortable with the integration of RDBMSs with Tomcat, and will be able to handle the most common requests for configuring RDBMSs to work with the Tomcat server.

JDBC Basics

JDBC is a programming interface for accessing RDBMSs. Its operation is based on the transmission and execution of Structured Query Language (SQL) on the database server. SQL is a text-based query language for performing operations with data stored in a relational database. In fact, JDBC is based on a Call-Level Interface (CLI) to an engine that processes SQL statements. More specifically, JDBC uses the X/Open SQL CLI (X/Open is an international standards organization) conforming to the SQL92 language syntax standard. Figure 13-1 illustrates how SQL CLI, and therefore JDBC, operates under the hood.

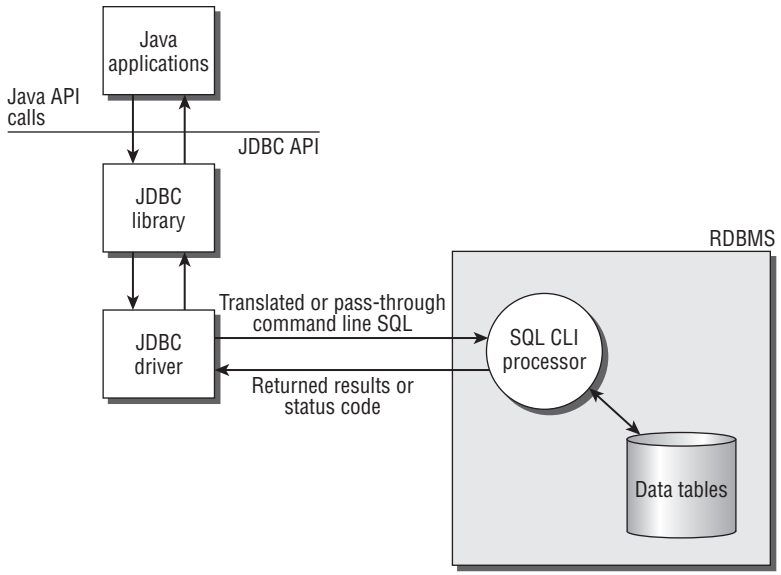

Figure 13-1: JDBC operation model

In Figure 13-1, the JDBC engine submits SQL query statements to the remote SQL processing engine (part of the RDBMS, it typically handles multiple simultaneous connections via a connection manager), and the SQL processing engine returns the result of the query in a set of data called a *result set*. A result set is typically zero or more rows of data. Think of result sets as a temporarily created table.

JDBC operations are designed to do the following:

- ❑ Take the JDBC API calls and transform them into an SQL query
- ❑ Submit that query to the SQL processing engine on the RDBMS
- ❑ Retrieve the result set that is returned from the query and transform it into a Java-accessible data structure

Not all statements return a result set. If a search is not successful, the returned result set will be empty (called a `NULL` result set). In addition, the SQL language includes statements that are used to create tables, update data, delete rows, and so on; these statements do not return any result sets either.

In JDBC programming, developers typically perform the following steps:

1. Obtain a connection to the remote database server (in JDBC 1.*x*, it is necessary to instantiate a database driver prior to obtaining a connection).
2. Create and prepare an SQL statement for execution (or call a stored procedure in the RDBMS).
3. Execute the SQL statement.
4. Obtain the returned result set (if any) and work on it.
5. Disconnect from the remote database.

Administrators are most interested in facilitating the first step — obtaining a connection to the desired database.

Establishing and Terminating Connections to RDBMSs

Other than providing a unified way of accessing, modifying, and manipulating data in RDBMSs, JDBC also provides a unified way to connect to RDBMSs from different vendors. While normal native connections to Oracle will be very different from connections to MySQL (which will be different from working with Microsoft's SQL server), connecting to any of these RDBMSs can be accomplished using the same JDBC API calls.

Evolving JDBC Versions

In the early days of JDBC, most Java developers were coding to the JDBC 1 standard. Under this standard, all code that needed to establish a connection to an RDBMS (as well as the code to disconnect from the RDBMS) was written by the developers. In fact, even the code to select and activate a JDBC driver was coded by the developers.

While simple and straightforward to code, this approach created a problem; in some cases where the driver used was hard-coded by the developers, the code to obtain a connection worked only with RDBMSs from a specific vendor.

With the arrival of JDBC 2, this restriction was relaxed. JDBC 2 introduced the concept of a *data source*. This is an indirect way of specifying the JDBC driver to be used for making the connection. Developers can now obtain a connection from the data source in their code, enabling the same JDBC code to work with drivers from any vendor. Meanwhile, an administrator can switch database vendor support by simply configuring a different data source, and no code changes are needed. The selection and configuration of data sources shifted from the developer to the administrator.

As Web applications became more complex, the demand for high-performance concurrent access to database connections increased. The code that developers write to maintain and share database connections becomes highly complex and error-prone. Because this code is utilitarian, and can be used by all applications, it is another area that JDBC 2 attempts to improve on (see the section "Database Connection Pooling," later in this chapter).

While JDBC 2's introduction of data source and connection pooling support opens up new possibilities for RDBMS developers, it falls short of specifying standard ways in which these features should (or must) be used. As a result, many of the architectural issues are left for the JDBC driver writers to

solve, and code can quickly become vendor-specific again (this time, depending on the JDBC driver vendor).

Furthermore, the rapid maturation of J2EE has consolidated its overall architecture. There is growing momentum to adopt the same resource adapter model to the Enterprise Information System (EIS) throughout the J2EE stack. Architecturally, JDBC connections are connections to external/legacy systems, which are considered part of the EIS. In the J2EE architecture, these connections should be managed through a well-defined Connector architecture. This is the subject matter of J2EE Connector Architecture (JCA). For more information, see the following URL:

```
http://java.sun.com/j2ee/connector/
```

In this architecture, J2EE software components access EIS resources via resource adapters with a common set of well-defined interfaces. These interfaces enforce well-defined contracts (between application server and resource adapter) in connection management, transaction management, and security. The evolution of the JDBC standard is migrating to this new JCA architecture as it becomes better defined.

The first step toward this migration is to detach any direct coupling between the application logic and the specific EIS resource that it needs. This can be accomplished by an intermediary indirect lookup mechanism. JNDI is a Java-based industry standard that can serve this purpose. JDBC 3 is the first version to be designed with this migration in mind.

In fact, JDBC 3 is the first specification that clearly spells out the different architectures that JDBC can operate in — including two-tier and three-tier models. The three-tier model corresponds to the application server model and is the model of operation favored by J2EE applications.

The JDBC specification also attempts to accommodate JDBC 1 and 2 drivers and model of operations, while formalizing JNDI as the preferred way for applications to obtain a data source. It also formalizes connection pooling as a value-added service of the application server or Servlet container.

JDBC 4, finalized in December of 2006, continues to support all features of the previous three versions. In addition, JDBC 4 addresses some features that make development easier, improve connection pooling, expose physical database row IDs for developers, and introduce a new XML data type to JDBC. With the introduction of Tomcat 6 in early 2007, JDBC 4 is still in its infancy in terms of vendor support.

Regardless of the JDBC version, the JDBC driver still must translate the unified JDBC commands into native commands to connect to the different servers. JDBC drivers have evolved significantly over the past few years and most of them today are high-performance Type IV drivers (explained in the next section). However, some legacy systems still exist that support only the older Type I to Type III drivers. It is a good idea to gain some familiarity with different types of JDBC drivers that may be around.

JDBC Driver Types

There are four different types of JDBC drivers: Type I to Type IV. In general, the higher driver types represent an improvement on architecture and performance, as described here:

- **Type I:** These drivers are the most primitive JDBC drivers, as they are essentially data access adapters. They adapt another data access mechanism (such as ODBC) to JDBC. These drivers completely rely on the other data access mechanism to work and, as such, create double the

administrative and maintenance headaches. These drivers are also typically hardware/OS-specific (because of the data access mechanism that they depend on), making them completely nonportable.

- **Type II:** These drivers are partially written in Java and partially written in native data access languages (typically C or C++). The non-Java portion of these drivers limits the portability of the final code and platform-migration possibilities. The administrative and maintenance burden of Type I still exists.
- **Type III:** These drivers are pure Java drivers on the client side, which gives them the portability benefit of Java. However, they rely on a middleware engine running externally to operate. The client code communicates with the middleware engine, and the engine talks to the different types of databases. The administration and maintenance burden is somewhat reduced, but far from eliminated.
- **Type IV:** These drivers are 100 percent Java and talk directly to the network protocols supported by the RDBMSs. This results in the highest performance connection and the most portable application code. Administration and maintenance are greatly simplified (only the driver needs to be updated).

Fortunately, most modern day JDBC drivers are of the Type IV variety. All the major RDBMSs available today (MySQL, Oracle, DB2, and MS SQL Server) have Type IV JDBC drivers available, either through the database vendors themselves or via a third-party driver vendor.

Database Connection Pooling

When a Web application accesses a remote RDBMS, it may do so through a *JDBC connection*. Typically, a physical JDBC connection is established between the client application and the RDBMS server via a TCP/IP connection. Establishing such a connection is CPU-, memory-, and execution time–intensive. It involves multiple layers of software, and the transmission and receipt of network data. A typical physical database connection may take several seconds to establish. Contrast this with the "cost" of doing a simple database query, which typically takes milliseconds to complete. It is obvious why it would be wise to decrease the number of connects and disconnects between queries.

Modern Web applications consist of JSPs and servlets that may need data from a database on every HTTP request (for example, a JSP that prints the current employees from a specific department, or an electronic auction system that enables you to see all your current open bids). On a well-loaded server, the time it takes to establish, disconnect, and reestablish actual connections (physical connections) can substantially slow down Web-application performance.

To create high-performance and scalable Web applications, JDBC driver vendors and application servers are incorporating database connection pooling into their products.

Connection pooling reduces expensive session establishment times (connects, disconnects, and reconnects) by creating a pool of physical connections when the system starts up. When an application requires a connection, one of these physical connections is provided. Typically, when an application finishes using a connection, it is disconnected. However, in the case of a logical connection, the associated physical connection is merely returned to the pool and awaits the next application request. Figure 13-2 illustrates database connection pooling.

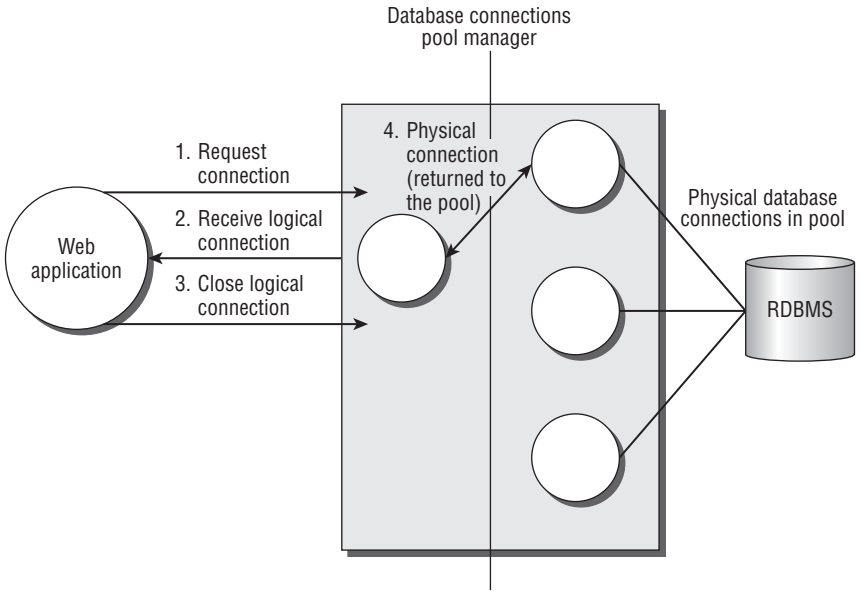

Figure 13-2: JDBC connection pooling

A *pool manager* creates the initial physical connections, manages the distribution of the physical connections to the Web applications in the form of a logical connection, returns any closed logical connection to the pool, and handles any exception or error that may arise — potentially disconnecting the physical connection or recovering from error conditions. Note that closing a logical connection does not actually close any physical connection, but merely returns the connection back to the pool. This pool manager functionality may be provided by one of the following three sources:

- An application server such as Tomcat 6
- A third-party pool manager software vendor
- The JDBC driver vendor

When configuring Web applications to run on Tomcat 6, the preferred and recommended pool manager to use is one that is supplied with the Tomcat server.

A Problem with Connection Pooling

Since the introduction of JDBC 3 and application server connections pooling in general, Tomcat users have gained some experience in the use of connection pools. Over time, connection pools can sometimes develop problems.

Since database connections are made over the network, they are not 100 percent reliable. Connections can sometimes fail in a way that you cannot recover from. Over time, since connections in a pool are kept open physically, many connections in the pool may start to go bad. The bad connections still hold up resources, and appear to be working to the pool manager. With JDBC 3 pooling, there is no reliable way of asking the connections if they are still valid. As a result, sometimes all connections in the pool need to be flushed, requiring a server restart.

JDBC 4 corrected this problem by providing a way for the pool manager to ask a connection if it is still valid. For this to work properly, it will require:

- The JDBC driver to implement this JDBC 4 feature, enabling a connection's validity to be queried
- The application server pool manager to support this feature — to query connection validity regularly to avoid having bad connections remaining in the pool

At the time of this writing, JDBC 4 specification has just been finalized, and implementations from commercial vendors or open source products are just starting to be available for early access.

Tomcat and the JDBC Evolution

Application developers and system designers using Tomcat 6 have a wide choice of JDBC support mechanisms from which to choose. Tomcat 6 provides JDBC 3 support while offering full backward compatibility to JDBC 2 (as well as JDBC 1). The remainder of this chapter examines the recommended mechanism to access JDBC resources while working with Tomcat, and explores one alternative access mechanism.

The major new JDBC features that are part of Tomcat 6 include the following:

- **Application server–managed database connection pools:** Tomcat 6 uses Jakarta Commons Database Connection Pooling (DBCP) to provide container-managed connection pooling. This also enables flexible configuration for the pooling mechanism (see the section "Resource and ResourceParams," later in this chapter). JDBC 3 is the first specification that defines standard configuration parameters for pooling (such as `maxStatements`, `initialPoolSize`, `minPoolSize`, `maxPoolSize`, and `maxIdleTime`), making the mechanism more configurable in a standards-compliant manner.
- **Using the JNDI-API to look up data sources within an application server:** Tomcat 6 emulates JNDI for Web applications running under it. This is a portable and configurable way of obtaining data sources for JDBC operations without hard-coding the driver and associated properties. It makes the selection of the JDBC driver and RDBMS instance a deferred deployment-time decision. JDBC 3 specifies JNDI as the preferred method for applications to locate a data source.
- **Ease of migration to the connector architecture:** Tomcat 6's decoupled architecture for access to JDBC data sources (through JNDI lookup) is a first step in the migration toward the JCA Connector–based architecture.

JNDI Emulation and Pooling in Tomcat 6

Tomcat provides valuable services for hosted Web applications that use JDBC connections. More specifically, Tomcat enables running Web applications to do the following:

- Access JDBC data sources using standard JNDI lookup
- Use connection pooling value-added service

The role of a Web-tier container as an intermediary between client Web applications and an RDBMS is defined by the set of J2EE specifications — most recently in the Servlet 2.4 specifications and the JDBC 3 specifications (located at `http://jcp.org/aboutJava/communityprocess/final/jsr154/` and `http://java.sun.com/products/jdbc/download.html#corespec30`). JDBC 4 specification is also at the same link, but Tomcat 6 support is forthcoming. The value-added service that Tomcat provides is compliant with these specifications, and is documented as a three-tier architecture (see Figure 13-3).

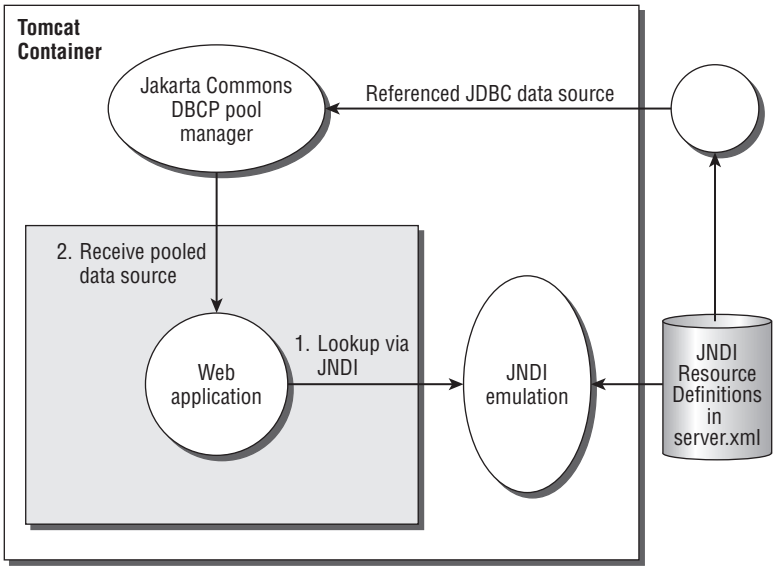

Figure 13-3: JNDI lookup of a JDBC data source in Tomcat 6

Figure 13-3 shows how JDBC drivers are configured with Tomcat as JNDI resources. These resources are made available during Web application runtime via standard JNDI lookups. The following steps are depicted in the diagram:

1. A Web application obtains a JNDI `InitialContext` from Tomcat; it then performs a lookup on the resource (JDBC data source) by name.
2. Tomcat handles the JNDI lookup by consulting the configuration files (`context.xml`, `server.xml`, and `web.xml`) to determine which JDBC driver to use for obtaining a data source. Tomcat can also use database connection pooling (DBCP) to pool the connections made; the connections obtained from Tomcat are logical connections.

Even though no true JNDI-compatible directory services are involved, the Tomcat container emulates the action of a JNDI provider. This enables code that is written with JNDI as the JDBC data source lookup mechanism to work within the Tomcat container (and other Servlet 2.4–compliant application servers).

As you can see in Figure 13-3, Tomcat 6 does more than merely provide JNDI emulation. It can also provide database connection pooling. Tomcat 6 uses *Apache Commons DBCP (database connection pooling)* for its built-in pool manager functionality.

Preferred Configuration: JNDI Resources

Using JNDI resources in Tomcat to configure JDBC data sources is the recommended way to provide Web applications with access to JDBC connections. While other methods are possible (and at least one alternative is discussed later in this chapter), this approach will result in code that is portable to other Web containers over the long term.

Following are the steps that must be followed to configure JNDI resources for a JDBC data source:

1. Add a `<Resource>` tag in the `<Context>` element (located in the `META-INF/context.xml` file) of the Web application, or in a `<DefaultContext>` subelement of the `<Host>` element (in the `server.xml` file) to configure the JNDI resource.
2. Ensure that a `<resource-ref>` element is defined, corresponding to the `<Resource>` from Step 1, in the `web.xml` file of the Web application using the JDBC resource.
3. Use JNDI calls in the application code to look up the JDBC data source.

The following sections provide more detail on how to perform each of these steps.

The Resource Tag

The `<Resource>` tag is used to specify the JNDI resource that represents a JDBC data source. Here is a typical `<Resource>` element found in the `server.xml` configuration file:

```
<Resource name="jdbc/WroxTC6" auth="Container"
   type="javax.sql.DataSource"
   maxActive="20"
   maxIdle="30"
   maxWait="10000"
   username="empro"
   password="empass"
   driverClassName="com.mysql.jdbc.Driver"
   url="jdbc:mysql://localhost:3306/wroxtomcat?autoReconnect=true"
/>
```

This resource statement has many attributes. The meaning of each of the attributes is described in the following table.

Attribute(s)	Description
`name="jdbc/WroxTC6"`	Create a JNDI resource that is accessible from the context (logical name) `java:comp/env/jdbc/WroxTC6` by the Web application. The `java:comp/env/` portion of the context is added on for all Tomcat-managed contexts. The Web application code can use this context to look up the data source.

Table continued on following page

Attribute(s)	Description
`auth`	Specifies if the Tomcat Container does the authentication on behalf of the application (`auth="Container"`) or if the application does it programmatically (`auth="Application"`).
`type="javax.sql.DataSource"`	The type of resource that will be returned during this lookup is a `javax.sql.DataSource`. It also specifies that the container should authenticate against the RDBMS on behalf of the Web application.
`maxActive`	Maximum number of active connections in the connection pool. A 0 value indicates unlimited.
`maxIdle`	Number of idle connections in pool before they are evicted. A –1 value indicates unlimited.
`maxWait`	Maximum number of milliseconds that the manager will wait for a database connection to respond, before throwing an exception. A –1 value indicates waiting indefinitely.

`username="empro"`
`password="empass"`
`driverClassName="com.mysql`
`.jdbc,Driver"`
`url="jdbc:mysql://localhost..."`
	The parameters for the MySQL Connector/J driver, including `username` and `password`. The `driverClassName` attribute tells Tomcat/DBCP the class to load as the JDBC driver. The URL is the MySQL-specific JDBC URL to access the database server instance.

You may want to check Chapter 6 for a detailed examination of the attributes allowed in the `<Resource>` element.

Working with Other RDBMSs

Tomcat's default DBCP factory will work with JDBC drivers for any RDBMSs. For example, here is the setting for accessing an Oracle database:

```
<Resource name="jdbc/WroxTC6"
          auth="Container"
          type="javax.sql.DataSource"
          driverClassName="oracle.jdbc.OracleDriver"
          url="jdbc:oracle:thin:@localhost:1521:wroxtomcat"
          username="empro"
          password="empass"
          maxActive="20"
          maxIdle="30"
          maxWait="-1"
/>
```

DBCP — Jakarta Commons Pooling Support

To return a JDBC data source to the application, Tomcat 6 uses a data source factory to create the data source. Tomcat 6 uses the Jakarta Commons DBCP (by default) to supply a data source factory and implement connection pooling.

Transactions and Distributed Transactions Support

RDBMSs offer varying levels of support for transactions. A transaction can be viewed as a unit of work that is composed of multiple operations, but can only be committed (all operations complete successfully) or rolled back (no operation completed). For example, MySQL 5 supports transactional access by default via the use of InnoDB tables.

When a transaction involves work that crosses multiple physical RDBMSs, it is called a *distributed transaction*. One standard that enables RDBMSs (and other products supporting transactions, such as Message Queue Servers) from different vendors to participate in the same distributed transaction is called XA.

In the XA operation model, an external *transaction manager* coordinates a two-phase-commit protocol between multiple *resource managers* (RDBMSs in this case). The two-phase-commit protocol ensures that the pieces of work, scattered across multiple physical RDBMSs, are either all completed or all rolled back.

JDBC 4, 3, and 2 all accommodate data sources that support XA operations. Administrators who work with XA data sources and data-source factories should consult the vendor's documentation for proper parameterization and ensure that they work with Tomcat.

Hands-On JNDI Resource Configuration

Now it is time to put theory into practice. The actual example presented here configures a DBCP data source, through Tomcat 6's JNDI resources support, with a Type IV JDBC driver. A JSP will be created that accesses the data in an actual RDBMS and displays it within a table on a generated HTML page. This example uses a popular, widely available RDBMS: MySQL.

Installation and configuration of the MySQL database is beyond the scope of this chapter. The discussion in this chapter assumes that you have MySQL already configured and tested, and that you have an account with privileges to create tables and add records to create the test database. The code in this chapter is tested against MySQL 5.0.27. The latest version of MySQL is available for download from the following URL:

```
mysql.com
```

The Type IV JDBC driver used here is the Connector/J, supplied by the MySQL vendor, which can be downloaded from the following URL:

```
mysql.org/downloads/connector/j/5.0.html
```

The latest version available as of this writing is 5.0.4, and is the version on which this example is based.

> *Note that you must unzip the driver JAR file from the download and use the* `mysql-connector-java-X. X. XX- bin.jar` *file within it. In this example, place this file under* `$CATALINA_HOME/lib`. *The example here is tested with MySQL Connector/J version 5.0.4, the latest stable version available as of this writing. However, any newer version should work identically.*

Creating the MySQL Test Database

To prepare for the example, you must create the MySQL database that will be used by the JSP. The employee database contains a number of employees from different branches of a company and details

about those employees. Assuming you have a MySQL user account that can create table privileges on a database called `wroxtomcat`, you can create the three tables.

If you have database system administrator privileges on MySQL, access can be granted via the `mysql` command line:

```
mysql> GRANT ALL PRIVILEGES ON wroxtomcat.* TO
    -> 'mike'@'localhost' IDENTIFIED BY 'abc123';
```

Follow this with a flush of the MySQL cache. This ensures that the change is immediately valid:

```
Mysql> FLUSH PRIVILEGES;
```

This enables the user `mike` to connect from `localhost` and create tables in the `wroxtomcat` database. The user `mike` must log on using the password `abc123`.

To make things easy, here is a `makedb.sql` script to create all the required tables:

```
CREATE DATABASE IF NOT EXISTS wroxtomcat;
USE wroxtomcat;
CREATE TABLE employee (
   employeeid VARCHAR(10) NOT NULL,
   name VARCHAR(50) NOT NULL,
   phone VARCHAR(15) NOT NULL,
   department VARCHAR(15) NOT NULL,
   password VARCHAR(15) NOT NULL,
   PRIMARY KEY (employeeid)
);
CREATE TABLE vacation (
   employeeid VARCHAR(10) NOT NULL,
   fiscal INT(3) NOT NULL,
   approved CHAR(1) NOT NULL,
   PRIMARY KEY (employeeid, fiscal)
);
CREATE TABLE dept (
   department VARCHAR(15) NOT NULL,
   address VARCHAR(30) NOT NULL,
   zipcode VARCHAR(6) NOT NULL,
   PRIMARY KEY (department)
);
```

Use `makedb.sql` to create the database, as follows:

```
$ mysql < makedb.sql
```

Next, load the tables with the data that will be used. This is performed via the SQL script `loaddb.sql`, which contains the following:

```
USE wroxtomcat;
INSERT INTO dept (department, address, zipcode) VALUES ( 'Engineering', '33
Mexicali Road', '25763');
```

```
INSERT INTO dept (department, address, zipcode) VALUES ( 'Sales', '15 Navel
Circle', '98322');
INSERT INTO dept (department, address, zipcode) VALUES ( 'Administration', '1
Lawless Court', '66699');
INSERT INTO employee (employeeid, name, phone, department, password) VALUES
( '2901', 'Joe', '333-3331', 'Engineering', 'junior');
INSERT INTO employee (employeeid, name, phone, department, password) VALUES
( '2202', 'Matt', '434-3333', 'Engineering', 'perlguru');
INSERT INTO employee (employeeid, name, phone, department, password) VALUES
( '3021', 'Jane', '231-0001', 'Sales', 'milseller');
INSERT INTO employee (employeeid, name, phone, department, password) VALUES
( '0001', 'Bill', '343-0012', 'Administration', 'gatorshaq');
INSERT INTO employee (employeeid, name, phone, department, password) VALUES
( '0015', 'Steve', '342-2212', 'Administration', 'billion');
INSERT INTO vacation (employeeid, fiscal, approved) VALUES ( '0001', '1',  'Y');
INSERT INTO vacation (employeeid, fiscal, approved) VALUES ( '0001', '2',  'Y');
INSERT INTO vacation (employeeid, fiscal, approved) VALUES ( '0001', '3',  'Y');
INSERT INTO vacation (employeeid, fiscal, approved) VALUES ( '0001', '4',  'Y');
INSERT INTO vacation (employeeid, fiscal, approved) VALUES ( '2901', '12', 'N');
INSERT INTO vacation (employeeid, fiscal, approved) VALUES ( '2202', '51', 'N');
```

This script simply fills the table with the data. Run the script from the MySQL console using the following command:

```
$ mysql < loaddb.sql
```

Now that the tables are created and you have populated them with data, you must create a user account that the developers will use to access the data within the database. Because the JSP functionality requires only read access to the data, creating a read-only user account for developer access is a good, secure practice. This ensures that data cannot be accidentally or maliciously modified or altered.

Setting Up the Read-Only User

If you do not have database system administrator privileges, you will need to seek help. You need the user setup shown in the following table.

User Property	Value
Username	`empro`
Password	`empass`
Access	`SELECT` only on the `wroxtomcat` database (that is, use the following command on the MySQL console: `GRANT SELECT ON wroxtomcat.* TO 'empro'@'localhost' IDENTIFIED BY 'empass';`)

This creates a user who has read-only access to the `wroxtomcat` tables, which the developer will be using to access the data in the table.

Adding the JDBC JNDI Resource to the Default Context

Finally, to configure the JNDI resource for the data source, follow the three-step approach outlined earlier and described in the following sections.

Step 1: Adding a context.xml with JNDI<Resource>

The first step is to make the JNDI data source accessible to the Web application that you are using. In this case, it is the `examples/jsp` application included with Tomcat 6. By adding the resource definition into its context (in the `META-INF/context.xml` file), the Web application can gain access to this resource.

In the `$CATALINA_HOME/webapps/examples` directory, create a `META-INF` directory if it does not yet exist. In this directory, create a `context.xml` application context descriptor, and place the following `<Context>` definition in this file:

```
<Context>
<Resource name="jdbc/WroxTC6"
      auth="Container"
      type="javax.sql.DataSource"
      maxActive="20"
      maxIdle="30"
      maxWait="10000"
      username="empro"
      password="empass"
     driverClassName="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost:3306/wroxtomcat?autoReconnect=true"
     />
</Context>
```

Two optional attributes (`removeAbandoned` and `removeAbandonedTimout`) help with the recycling of database connections, even for those that may not be released properly because of faulty developer coding. When the number of available connections in the pool runs low, the DBCP pool management code will recycle connections based on an elapsed idle timeout. For example, the following additional parameters will cause the DBCP pool management code to recycle all JDBC connections that are idled for more than 2 minutes (120 seconds):

```
<Resource ...
  removeAbandoned="true"
  removeAbandonedTimeout="120"
 ...  />
```

Tomcat 6 includes the required Jakarta Commons libraries by default; you will find the DBCP and dependent libraries all packaged in the `$CATALINA_HOME/lib/tomcat-dbcp.jar` file.

Step 2: Adding the <resource-ref/> Entries to web.xml

Instead of creating a new Web application, an easy way to add a test JSP is to add it to an existing example application from Tomcat. To do this, change the directory to `$CATALINA_HOME/webapps/examples/WEB-INF` and edit the `web.xml` file (this is the deployment descriptor of the `jsp-examples` Web application). Add the following highlighted code to `web.xml` (note that it should be added immediately after the last `<env-entry>` element in the file, very close to the end of the file):

```
...
<env-entry>
      <env-entry-name>foo/name4</env-entry-name>
      <env-entry-type>java.lang.Integer</env-entry-type>
      <env-entry-value>10</env-entry-value>
</env-entry>
```

```
<resource-ref>
 <res-ref-name>jdbc/WroxTC6</res-ref-name>
 <res-type>javax.sql.DataSource</res-type>
 <res-auth>Container</res-auth>
</resource-ref>
```

This `<resource-ref>` entry makes the `jdbc/WroxTC6` context, via JNDI APIs, available within the `jsp-examples` Web application.

Step 3: Using JNDI to Look Up a Data Source

Finally, it is time to write the code that will look up the data source and start querying the database. The following JSP, `JDBCTest.jsp`, will do exactly this. Put it into a `$CATALINA_HOME/webapps/examples/jsp/wroxjdbc` directory (create this directory yourself). Pay special attention to the way JNDI is used to obtain the data source (JSTL code highlighted).

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page errorPage="errorpg.jsp" %>
<html>
  <head>
    <sql:query var="employees" dataSource="jdbc/WroxTC6">
        select * from employee;
    </sql:query>
  </head>
  <body>
    <h1>JDBC JNDI Resource Test</h1>
    <table width='600Ð border='1Ð>
      <tr>
        <th align='left'>Employee ID</th>
        <th align='left'>Name</th>
        <th align='left'>Department</th>
      </tr>
<c:forEach var="employee" items="${employees.rows}">
        <tr>
          <td> ${employee.employeeid}</td>
          <td> ${employee.name} </td>
          <td> ${employee.department} </td>
        </tr>
</c:forEach>
    </table>
  </body>
</html>
```

JSTL is a JSP Standard Tag Library, a utility library enabling JSP to create pages without embedding Java code. Instead of code, JSP developers simply include tags in the page and parameterize them. One very useful set of tags provides JDBC access through a JNDI data source. This set of JSTL SQL tags is used within the example.

The code highlighted here looks up the JNDI resource that is configured as `jdbc/WroxTC6`. The `<sql:query>` tag is then used to perform a `SELECT *` on the `employee` table. The rows are returned in an `employees` variable. A `<c:forEach>` iteration tag is used to print out all the rows that are retrieved.

Within the loop, field values within the HTML table are rendered using EL expressions (EL is Expression Language, a built-in feature of JSP 2.1).

Any exception caught during execution of this JSP is redirected to a very simple error-handling page called the `errorpg.jsp` file. This file is specified via the `errorPage` attribute of the `@page` directive. Here is the content of `errorpg.jsp`:

```
<html>
  <body>
    <%@ page isErrorPage="true" %>
    <h1> An error has occurred </h1>
    <%= exception.getMessage() %>
  </body>
</html>
```

This page simply displays the detailed error message describing the exception thrown.

Testing the JNDI Resource Configuration

At this point, the JNDI resources are prepared, the database tables are populated, the `<resource-ref>` to the deployment descriptor has been added, and a JSP that will use JNDI to obtain a JDBC data source has been created. It is time to give the new JSP code a test.

Start Tomcat 6 and then, from a browser, attempt to reach the following URL:

```
http://localhost:8080/examples/jsp/wroxjdbc/JDBCTest.jsp
```

This will compile and execute the JSP code. If everything is configured correctly and working, your browser display should be similar to what is shown in Figure 13-4.

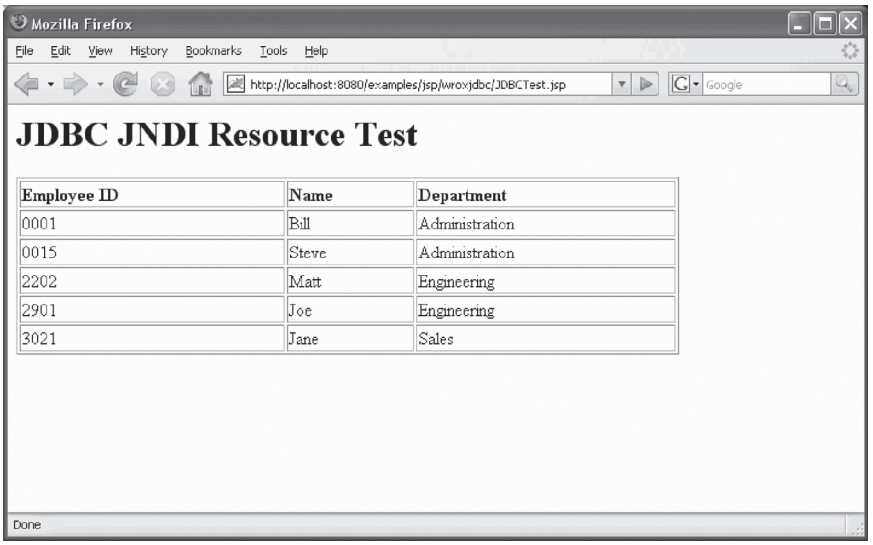

Figure 13-4: Display of MySQL RDBMS data in a JSP

The Web page shown in Figure 13-4 is the result of a JDBC query to the MySQL database data, via a connection obtained from the JNDI lookup.

You may face exceptions such as the server denying access to the data source, or some server connection failures, which are caused by the MySQL user account not having enough privileges.

The Error Page in Action

To see the error page exception, first shut down the MySQL server. This causes an exception to be thrown after several attempts are made to connect to it.

Now try to access the URL of the `JDBCTest.jsp`:

```
http://localhost:8080/examples/jsp/wroxjdbc/JDBCTest.jsp
```

This time, the page does not render immediately. It takes a little while. Then, a page will be displayed that is rendered using the `errorpg.jsp` error page. This is shown in Figure 13-5.

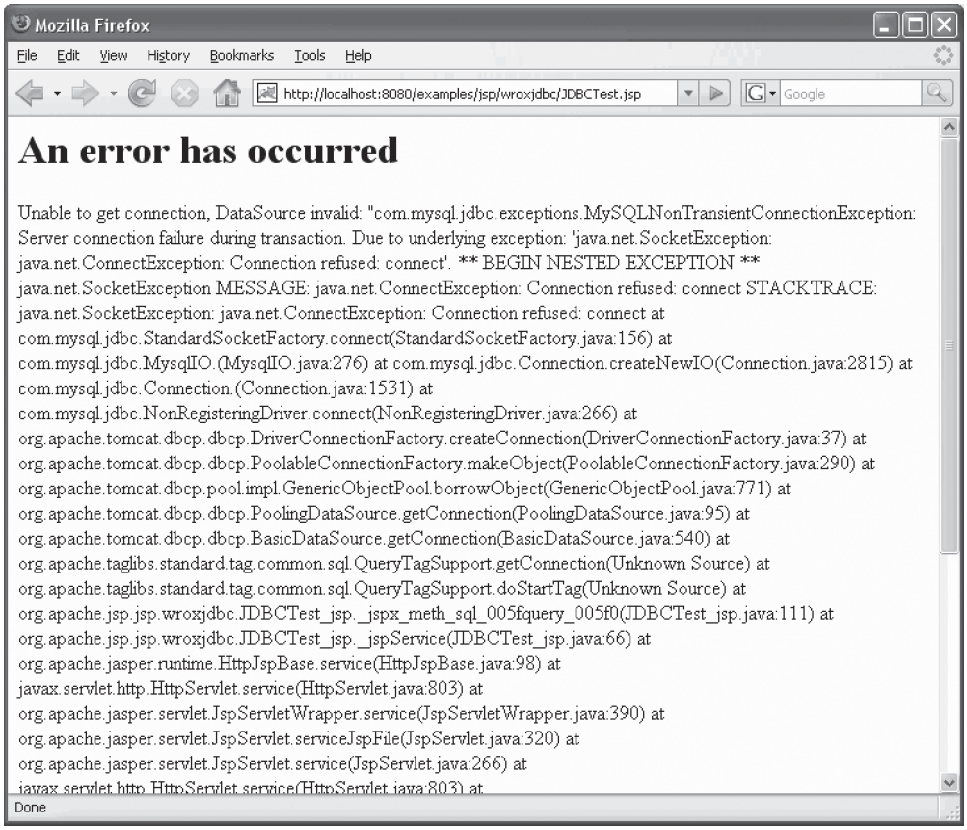
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/examples/jsp/wroxjdbc/JDBCTest.jsp
Google
An error has occurred
Unable to get connection, DataSource invalid: "com.mysql.jdbc.exceptions.MySQLNonTransientConnectionException: Server connection failure during transaction. Due to underlying exception: 'java.net.SocketException: java.net.ConnectException: Connection refused: connect'. ** BEGIN NESTED EXCEPTION ** java.net.SocketException MESSAGE: java.net.ConnectException: Connection refused: connect STACKTRACE: java.net.SocketException: java.net.ConnectException: Connection refused: connect at com.mysql.jdbc.StandardSocketFactory.connect(StandardSocketFactory.java:156) at com.mysql.jdbc.MysqlIO.(MysqlIO.java:276) at com.mysql.jdbc.Connection.createNewIO(Connection.java:2815) at com.mysql.jdbc.Connection.(Connection.java:1531) at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:266) at org.apache.tomcat.dbcp.dbcp.DriverConnectionFactory.createConnection(DriverConnectionFactory.java:37) at org.apache.tomcat.dbcp.dbcp.PoolableConnectionFactory.makeObject(PoolableConnectionFactory.java:290) at org.apache.tomcat.dbcp.pool.impl.GenericObjectPool.borrowObject(GenericObjectPool.java:771) at org.apache.tomcat.dbcp.dbcp.PoolingDataSource.getConnection(PoolingDataSource.java:95) at org.apache.tomcat.dbcp.dbcp.BasicDataSource.getConnection(BasicDataSource.java:540) at org.apache.taglibs.standard.tag.common.sql.QueryTagSupport.getConnection(Unknown Source) at org.apache.taglibs.standard.tag.common.sql.QueryTagSupport.doStartTag(Unknown Source) at org.apache.jsp.jsp.wroxjdbc.JDBCTest_jsp._jspx_meth_sql_005fquery_005f0(JDBCTest_jsp.java:111) at org.apache.jsp.jsp.wroxjdbc.JDBCTest_jsp._jspService(JDBCTest_jsp.java:66) at org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:98) at javax.servlet.http.HttpServlet.service(HttpServlet.java:803) at org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:390) at org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:320) at org.apache.jasper.servlet.JspServlet.service(JspServlet.java:266) at javax.servlet.http.HttpServlet.service(HttpServlet.java:803) at
Done

Figure 13-5: The error handling JSP page showing an Exception Stack Trace

If you look to the very end of this page, you can find the root cause of the problem. The server has tried three times to connect to the MySQL server and failed.

Alternative JDBC Configuration

In Tomcat 6, the JNDI API is the preferred and recommended way to pass a JDBC data source to your Web applications. It is currently the best-supported mechanism for accessing JDBC data sources.

However, in production, you might encounter situations in which you have to consider alternative means of JDBC data source or connection access.

Typically, there is no reason why newly developed database access code slated for Web application deployment should not use the preferred JNDI mechanism for data access. However, because JDBC 1 was a widely used and highly functional API long before the arrival of JDBC 2, 3, or 4, a large legacy base of working JDBC code remains unaware of data sources and connection pooling.

In some circumstances, you must integrate legacy code, and the source code is either not available or cannot be changed. Typically, legacy JDBC 1 code has both the JDBC driver and the URL of the database hard-coded. Thankfully, newer versions of JDBC continue to maintain backward compatibility with JDBC 1. This means that legacy code can continue to run, even in Tomcat 6 servers.

Another potential reason for deviating from the recommended configuration is the deployment of an alternative connection pool manager. This could happen, for example, with a shared hosting ISP using a commercial product that does not support the Jakarta Commons DBCP pooling. In addition, developers sometimes disagree about the merits of one pool manager implementation over another.

Alternative Connection Pool Managers

Up until Tomcat 4.1.*x*, connection-pooling implementation on Tomcat servers evolved in a roll-your-own manner. Because Tomcat did not provide default support, anyone who needed the functionality (which included anyone who deployed any large- or medium-scale Web application) had to write their own code, or find a third-party solution to the problem.

As a result, it is quite likely that some legacy projects on Tomcat are using an alternative connection-pooling manager, with its own requirements for data-source configuration.

The following section examines one such pool manager, and shows how its configuration and access differ from the preferred method. The alternate pool manager is c3p0. It is an open-source project that is hosted on SourceForge at the following URL:

```
http://sourceforge.net/projects/c3p0/
```

About the c3p0 Pool Manager

Since before the working release of DBCP, the open-source c3p0 provided flexible object pooling for developers. c3p0 provides a wrapper for legacy JDBC drivers with DriverManager-based interfaces, and provides them with a JNDI DataSource–based API. c3p0 0.9.1 (the latest version available as of this writing) provides the following features:

- DriverManager-based (JDBC 1 or 2) JDBC drivers to a new DataSource-based interface
- Pooling of connections for legacy drivers

- Compatibility with most JNDI naming services
- Easy use for developers; all classes in a single JAR
- Tolerance for developers' quirks when closing connections, statements, and result sets
- Functionality in a hosted environment, on a per-application basis, without affecting other Web applications running on the same server
- Separate pools for connections with differing authentication credentials
- Additional support pooling of JDBC PreparedStatements (speeds up frequently used queries)

Some of these features may be attractive to developers, and may indeed be the reason why cp30 is deployed in specific production scenarios.

The choice of one pool management strategy over another is highly dependent on the application, the system configuration, the data-access pattern, and subjective designer/developer preferences.

Deploying the c3p0 Pooling Manager

The c3p0 binaries are packaged as a single JAR file, called `c3p0-x.x.x.jar` where `x.x.x` is the version number. As of this writing, the latest available version is 0.9.1 and all code in this section has been tested against 0.9.1.

The placement of this library in the Tomcat directory hierarchy will depend on your specific pool-management strategy. For example, if you were to pool JDBC connections on a per-Web-application basis, the `c3p0-x.x.x.jar` file should be placed under the `WEB-INF/lib` directory of your Web application. The following example assumes that you will be using c3p0 for only a single Web application (and not server-wide; for server-side deployment, you need to place it into the `%CATALINA_HOME%/lib` directory).

c3p0 supports its own XML-based configuration file. The XML configuration file allows the definition of multiple named configurations that can be programmatically selected at runtime. Instead of the XML configuration file, a properties file can be used.

However, all of the configurable attributes can also be set programmatically. The examples in this chapter use programmatic configuration. Please see the c3p0 documentation for configuration file details.

Obtaining JDBC Connections Without JNDI Lookup

c3p0 includes support of connection pooling for JDBC 1 drivers, before JDBC 2 became popular. In those times, it was necessary to hard-code the JDBC driver instantiation and connection establishment right into the JSP — JNDI support is not available.

To illustrate how different such an approach may look, this first example obtains a JDBC connection directly, after wrapping cp30's `ComboPooledDataSource` around the JDBC driver.

The JSP that you create is placed in the `%CATALINA_HOME%/webapps/examples/jsp/wroxjdbc` directory. The file is called `C3P0Test.jsp`. The code is listed here, with the hard-coded JDBC portion highlighted:

```
<h1>JDBC Test - Direct C3P0</h1>
<%!
    private static ComboPooledDataSource pool;
    public void jspInit() {
    try {
      pool = new ComboPooledDataSource();
      pool.setDriverClass( "com.mysql.jdbc.Driver" );
      pool.setJdbcUrl( "jdbc:mysql://localhost:3306/wroxtomcat?autoReconnect=true" );
      pool.setUser("empro");
      pool.setPassword("empass");
      pool.setAcquireIncrement(3);
      pool.setMaxPoolSize(30);
    } catch (Exception ex) {
    // JSP init() cannot throw any exception ex.printStackTrace();
  }
 }
%>
```

By overriding `jspInit()`, a static instance of `ComboPooledDataSource` is created the first time `CP30Test.jsp` is executed. This instance is named `pool`, and contains your connection pool. The following highlighted code shows how to use the instance to get a JDBC connection from the pool.

```
    <%
    Connection conn= pool.getConnection();
    Statement stmt= conn.createStatement();
    ResultSet rset= stmt.executeQuery("select * from employee;");
    %>
    <table width='600' border='1'>
      <tr>
        <th align='left'>Employee ID</th>
        <th align='left'>Name</th>
        <th align='left'>Department</th>
      </tr>
      <%
      while (rset.next()) {
      %>
        <tr><td> <%= rset.getString(1)    %></td>
          <td> <%= rset.getString(2)  %></td>
          <td> <%= rset.getString(4)  %></td>
        </tr>
      <%  }
      if (stmt != null)
        stmt.close();
      conn.close();
      %>
    </table>
  </body>
</html>
```

The highlighted explicit call to `conn.close()` ensures that the connection is returned to the pool. Note that once the connection to the database is obtained, the code is almost identical to that of `JDBCTest.jsp`. While this is backwardly compatible with JDBC 1, and supported by Tomcat 6, it is not a recommended practice. The next example shows how to integrate c3p0 with Tomcat 6 through Tomcat's JNDI emulation.

Testing Non-JNDI Pool Access with c3p0

To test the non-JNDI method of obtaining JDBC connections using cp30, start Tomcat 6 and try to access the JSP via the following URL:

```
http://localhost:8080/examples/jsp/wroxjdbc/CP30Test.jsp
```

The screen that you see should display data from the `employee` table, shown in Figure 13-6. This is identical to the result from `JDBCTest.jsp`, where the default DBCP pooling was used. The first time the preceding URL is accessed, Tomcat 6 will compile the JSP and call `jspInit()`, creating a static `ComboPooledDataSource` instance. The core JSP code obtains a pooled connection from this data source and uses it to query the `employee` table.

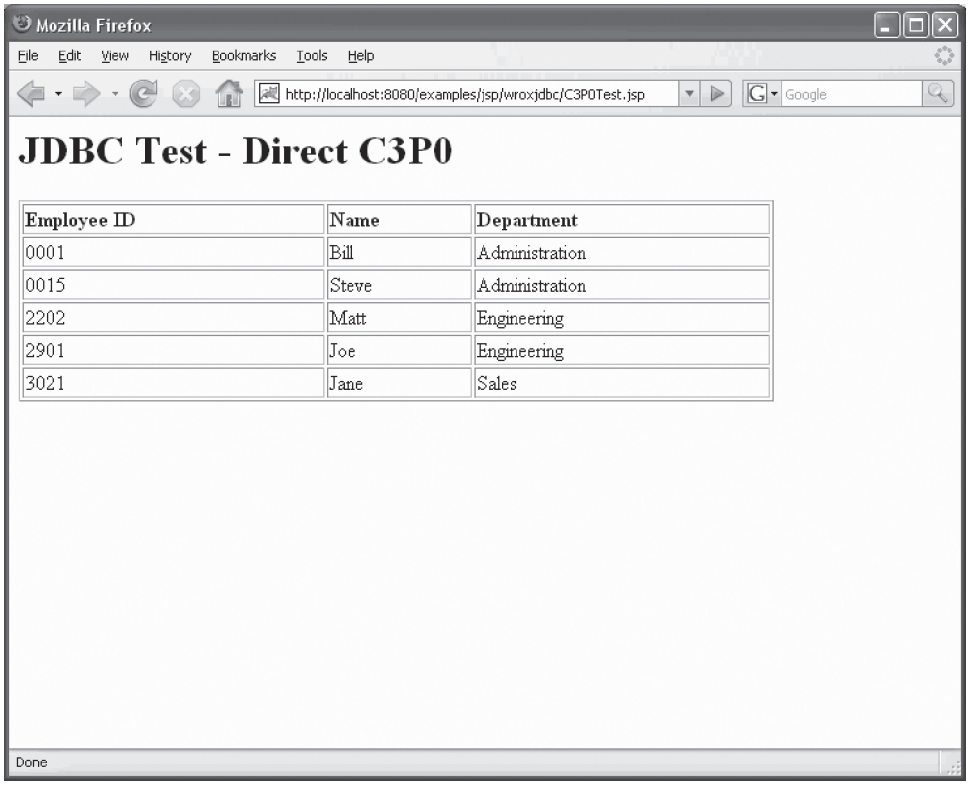

Figure 13-6: JSP-generated output from a direct C3P0–based JDBC test

Obtaining a Connection with JNDI Mapping

The c3p0 open-source project provides JNDI support when locating a JDBC data source. The real good news is that Tomcat 6–styled JNDI emulation is directly supported. You do not have to write any extra Java adapter code to use c3p0 instead of the DBCP pool manager.

All that is needed for compatibility with Tomcat 6 is an object factory class to hand out the required data source, and c3p0 comes with such a factory class. The `ComboPooledDataSource` can be used together with `org.apache.naming.factory.BeanFactory` to make this happen.

You can use this approach to replace Tomcat's DBCP pool manager with c3p0 as an alternative. This will enable you to maintain JNDI-based access, keeping the application code highly portable. To use this approach, make sure you already have both the cp30 JAR file and the Connector/J JAR file in the application's lib directory, at `%CATALINA_HOME%/webapps/examples/WEB-INF/lib`.

To enable the access of c3p0 through Tomcat 6's JNDI, it is necessary to edit the `%CATALINA_HOME%/webapps/examples/META-INF/context.xml` file. Within the `<Context>` element, add the following resource declaration for the `jdbc/wroxtomc` resource:

```
    <Context>
    <Resource     name="jdbc/WroxC3P0"
                  auth="Container"
                  driverClass="com.mysql.jdbc.Driver"
                  maxPoolSize="30"
                  minPoolSize="3"
                  acquireIncrement="3"
                  user="empro"
                  password="empass"
                  factory="org.apache.naming.factory.BeanFactory"
                  type="com.mchange.v2.c3p0.ComboPooledDataSource"

    jdbcUrl="jdbc:mysql://localhost:3306/wroxtomcat?autoReconnect=true" />
     <Resource name="jdbc/WroxTC6"
               auth="Container"
               ...
```

The `factory` attribute overrides the default factory of `org.apache.commons.dbcp.BasicDataSourceFactory`. Instead, it is using the generic BeanFactory (in `org.apache.naming.factory.BeanFactory`), which will create an instance of cp30-managed data sources as `ComboPooledDataSource`.

To place a test JSP into the `examples` Web application, you must edit the deployment descriptor (`web.xml`) and add a reference `<resource-ref>` to the preceding resource at the end of the file:

```
    <resource-ref>
      <res-ref-name>jdbc/WroxC3P0</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>
```

These configuration steps are identical to what was necessary when DBCP was used earlier in the setup of `JDBCTest.jsp`. In general, this would be the way to configure any alternative pooling mechanism that is compatible with Tomcat 6's JNDI emulation.

Last but not least, a test JSP file must be created that makes use of the `jdbc/WroxC3P0` JNDI resource. To create this JSP file, it is necessary to make only a very small change to `JDBCTest.jsp`. The change required is highlighted in the following snippet, and the completed file is placed into the `jsp/wroxjdbc` subdirectory of the `examples` Web application. The new JSP is called `C3P0Test2.jsp`.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page errorPage="errorpg.jsp" %>
<html>
  <head>
<sql:query var="employees" dataSource="jdbc/WroxC3P0">
select * from employee;
</sql:query>
  </head>
  <body>
    <h1>JDBC JNDI Resource Test Using C3P0</h1>
    <table width='600Ð border='1'>
    ...
```

Note that the code of `JDBCTest.jsp` and `C3P0Test2.jsp` is essentially identical. This demonstrates the advantage of decoupling the application code from the RDBMS accessed via JNDI. The same code can be used with connection pooling mechanisms from different vendors without change; only the configuration must be modified.

Testing c3p0 with Tomcat 6 JNDI-Compatible Lookup

To test c3p0's JNDI-compatible operation mode, shut down Tomcat 6 and restart it. This restart is not strictly necessary. However, restarting Tomcat ensures that you begin with a clean slate, as the previous example loaded c3p0. Next, try to access the JSP via the following URL:

```
http://localhost:8080/examples/jsp/wroxjdbc/C3P0Test2.jsp
```

The output from this example is again similar to what is shown in Figure 13-6.

Note that the output is indistinguishable (other than the new title) from that of `JDBCTest.jsp` or `C3P0Test2.jsp`, although they each use distinctively different pooling code to access and manage the JDBC connection used.

> *As of this writing (in early 2007), the c3p0 project is alive and well on SourceForge, which has the latest code updates and bug fixes. Previous versions of the book used a different and now defunct pool manager. This does bring up the issue of ongoing support for anyone who may decide to adopt an alternative pool manager.*

Deploying Third-Party Pools

Having explored the issues surrounding the integration of third-party pool managers, carefully consider the consequences of doing so before proceeding.

Following are two main points that must be considered:

- **Support:** How well is the third-party pool manager supported? If there is any future incompatibility with Tomcat, who will resolve it and how soon?
- **Code portability:** Must one sacrifice configuration flexibility when using the pool manager? Is it necessary to hard-code driver and data source information?

Because DBCP is an Apache Commons project, it is used by many Apache Jakarta projects. As such, it is likely to evolve and stabilize rapidly with contributions from the Jakarta community. Third-party connection pool managers are unlikely to enjoy the same level of contribution and support.

In addition, because it is an essential and integral part of Tomcat 6, DBCP technology will track Tomcat evolution and will always be tested for compatibility with every new Tomcat release.

Even if a production scenario forces the deployment of a nonstandard pool manager, it is wise to consider a gradual migration to standard DBCP. This is especially true if Tomcat 6 deployment is important.

Summary

This chapter explored JDBC connectivity in the context of Tomcat 6. The most obvious interaction is the need for Web applications to connect to relational database sources.

To conclude this chapter, let's review some of the key points that we discussed:

- Java supports the accessing of RDBMSs in the form of JDBC. We examined the evolution of JDBC versions, including coverage of each type of JDBC driver that is available.
- With the latest Servlet 2.4 and JDBC 3 and 4 standards, the recommended way of providing a JDBC data source to Web applications involves the configuration of JNDI resources in the Tomcat configuration file. In addition, Tomcat 6 provides a value-added database connection pooling service. This pooling service draws on the code from the Jakarta Commons DBCP project.
- Using the latest MySQL and its Connector/J driver, JNDI resources can be configured for a custom JSP that accesses RDBMS tables to generate a dynamic HTML page.
- Database connection pooling is required functionality for any serious Web application, but standard connection pooling on Tomcat is a relatively recent phenomenon. As a result, many current third-party solutions are not standards-compliant. As administrators, it is important to realize the existence of such alternatives. In fact, many legacy systems today still deploy them, and some shared hosting ISPs require the use of them.

- A third-party pool manager called cp30 can be deployed with Tomcat 6. This pool manager replaces the built-in DBCP-based connection pool. c3p0 can use a legacy JDBC 1 mechanism for providing pooled JDBC connection access to a Web application. As a result, the Web application coding becomes specific to the pool manager, despite the fact that most of the application code can be reused. This raises serious questions about the actual value of such a third-party pool manager. However, this pool manager is also "Tomcat 6 JNDI-compatible" By using the JNDI-compatible configuration, the Web application coding remains portable, and independent of the pool manager mechanism used. This example illustrates that the flexible JNDI-based data source distribution is not limited to Tomcat 6's built-in DBCP-based pool manager, but can be leveraged by any third-party pool manager as well.

The next chapter examines Tomcat security.

14

Tomcat Security

Perhaps no topic in the computing industry receives more emphasis than security, and for good reason. As network computing enters the twenty-first century, it is clearer than ever that the Internet is not a safe place. Attacks can be simple pranks (such as defacing a Web site), or take much more serious forms, such as industrial espionage, sabotage, or the theft of consumer information. System administrators must take many steps to secure network-exposed systems and services (such as Tomcat) against such aggressions.

This chapter covers topics relating directly to the security of your Tomcat server and applications running on it, including:

- ❏ Verifying initial download integrity
- ❏ Securing Tomcat against common attacks
- ❏ Running Tomcat with an unprivileged user account
- ❏ Locking down the file system
- ❏ Limiting access to Web applications with authentication Realms
- ❏ Turning off `DefaultServlet` directory listing capability
- ❏ Guarding against default `web.xml` configuration vulnerability
- ❏ Encrypting communications between Tomcat and application clients with SSL

The discussion of security issues surrounding the Tomcat server and applications is not entirely platform-agnostic. However, this chapter does not attempt to provide platform-specific instructions for all operating systems. Where appropriate, specific instructions are provided for Windows 2003/XP and Linux operating systems. Despite some pockets of platform-specificity, the principles shared in this chapter are applicable to any secure operating system. This chapter does not cover other production network security issues (such as the configuration of security firewalls, DMZ setup, and so on) not directly related to Tomcat configuration.

Verifying Tomcat Download Integrity

If the Tomcat binaries that you've downloaded have been modified by a malicious source, there is no limit to the damage and information loss that may result. No amount of additional security actions can be effective if you start out with changed server binaries.

Open source software is great for developers because the source code is available and easily modified to include custom features or requirements. On the flip side, malicious hackers can also take advantage of this easy modification to introduce Trojan code, malware, or spyware modules.

The probability for tampering is increased by the fact that most open source software is downloadable from many mirrored Web sites — other than the originating group. Chances are that you have downloaded the Tomcat server binaries from a mirror site — a machine hosted by groups that you may not be familiar with.

Therefore, to ensure a secure installation, the first thing you must do is to verify that the binaries that you will execute in production are indeed, bit for bit, the binaries that has been released by the developers at Apache.

Thankfully, verifying the integrity of your downloaded Tomcat 6 is straightforward. Verification can be performed through a manual checksum and a set of PGP keys.

> *Important: To ensure that the following process is valid, you must be certain that you are operating in a network that can be depended on. Because you are accessing* `tomcat.apache.org` *and* `apache.org` *to get all your authoritative PGP signature and checksums, you must be certain that the DNS (Domain Name Service) server is managed by a trusted source and that your local machine that you run your browser on is fully secured. Some existing exploits direct you to a malicious host instead of to the official site, such as* `apache.org`*.*

Verifying the MD5 DIGEST

One way to ensure the integrity of the Tomcat download is to compare their MD5 digests. MD5 digests are computed by scanning and operating on every bit of the file using a digesting mechanism. Most MD5 utilities use the 128-bit MD5 algorithm created by Ron Rivest from MIT and RSA Data Security, Inc. This algorithm is also the subject matter of Internet RFC 1321. See the "DIGEST" subsection of the "Authentication Mechanisms" section later in this chapter for more information on digest algorithms in general.

The algorithm is sensitive to the bytes in the file, as well as the ordering of those bytes. All Tomcat downloads include an official MD5 digest. After downloading the file, you can use an MD5 utility to compute the file's MD5 digest, and then compare it against the officially published digest.

Even changing one single bit in the file will result in a different MD5 digest. If the two MD5 digests are exactly equal, you can have confidence that the file you've downloaded has not been tampered with. If you're using Linux, you should already have an MD5 utility installed. If you are using Windows, various MD5 digest utilities are available, and you can find a copy of a command-line MD5 utility at the following site:

```
http://www.fourmilab.ch/md5
```

Figure 14-1 shows the Tomcat download page, with the link to the MD5 digest of files.

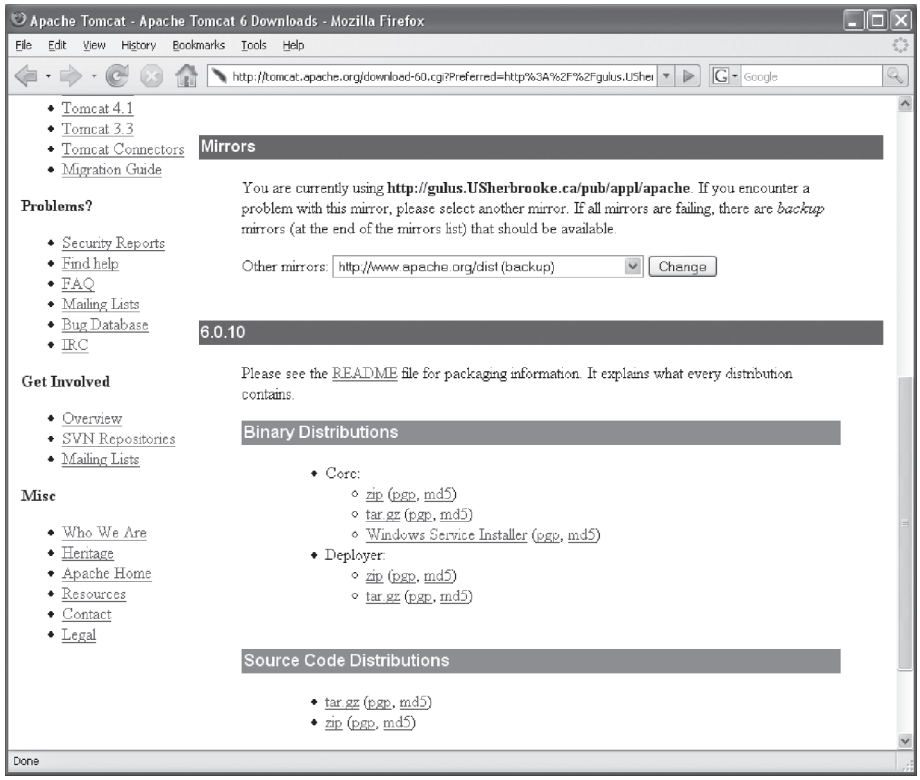

Figure 14-1: The PGP and MD5 digests available for download

If you click the associated MD5 link, the official MD5 digest will be displayed. You need to generate your own MD5 and compare it against the official digest.

Figure 14-2 shows a run of the MD5 utility for the `apache-tomcat-6.0.10.zip` download.

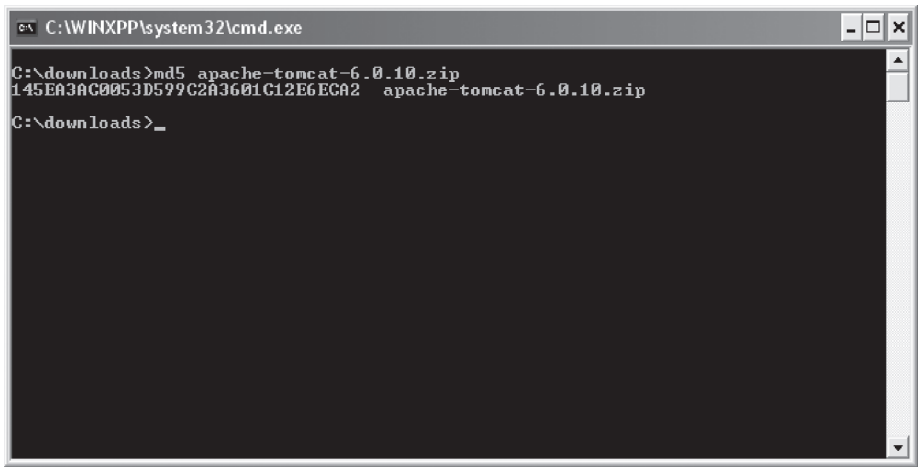

Figure 14-2: Computing MD5 for downloaded binary

In the case of Figure 14-2, the computed message digest, `145ea3ac0053d599c2a3601c12e6eca2`, matches the official one exactly.

Other than verifying the MD5 digest, you can also verify the server download using PGP signatures.

Using PGP to Verify the Download

As an alternative to, or as an addition to, MD5 verification, you can also use PGP. PGP (Pretty Good Privacy) is a cryptographic encryption/decryption utility developed by Phil Zimmermann. It is widely used for ensuring privacy and for authentication. The protocol used by PGP is called OpenPGP and is described by the Internet RFC 2440.

On Linux systems, you probably already have an implementation of OpenPGP — the GNU Privacy Guard (gpg) — installed. If not, you can use the operating system's software update service to obtain the binary. On Windows systems, you need to obtain an implementation on your own.

PGP Corporation (`pgp.org`) provides a comprehensive suite of security software based on this technology. Currently the trial client software download reverts to a limited featured freeware after the trial period. Alternatively, you can find binaries of the command-line utility for Windows at:

```
gnupg.org/download/
```

If you obtain binaries from the preceding link, you need to add the installation directory of the binaries to your PATH environment variable before they are available from the command line.

To confirm proper installation and accessibility, use the following command:

```
gpg --help
```

This prints a page of help information, similar to the following:

```
C:\Program Files\GNU\GnuPG>gpg --help
gpg (GnuPG) 1.4.6
Copyright (C) 2006 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.
Home: C:/Documents and Settings/Sing/Application Data/gnupg
Supported algorithms:
Pubkey: RSA, RSA-E, RSA-S, ELG-E, DSA
Cipher: 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH
Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compression: Uncompressed, ZIP, ZLIB, BZIP2
Syntax: gpg [options] [files]
sign, check, encrypt or decrypt
default operation depends on the input data
Commands:
 -s, --sign [file]             make a signature
     --clearsign [file]        make a clear text signature
 -b, --detach-sign             make a detached signature
 -e, --encrypt                 encrypt data
 -c, --symmetric               encryption only with symmetric cipher
```

```
-d, --decrypt                decrypt data (default)
    --verify                 verify a signature
    --list-keys              list keys
    --list-sigs              list keys and signatures
    --check-sigs             list and check key signatures
    --fingerprint            list keys and fingerprints
-K, --list-secret-keys       list secret keys
    --gen-key                generate a new key pair
    --delete-keys            remove keys from the public keyring
```

After you have the client software, you can download and import the public PGP keys of the release managers for Tomcat. This is typically available from the official Tomcat site, in a `KEYS` file. See Figure 14-3 for an example.

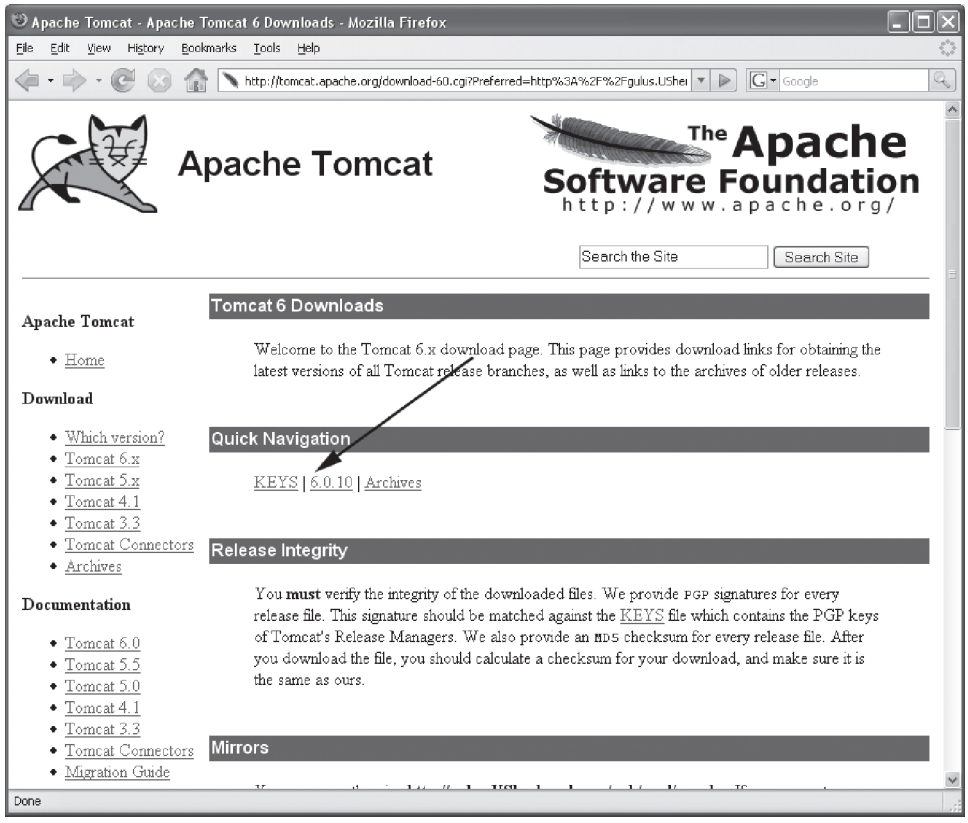
Apache Tomcat - Apache Tomcat 6 Downloads - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Apache Tomcat
The Apache Software Foundation
http://www.apache.org/
Search the Site
Search Site
Apache Tomcat
Home
Download
Which version?
Tomcat 6.x
Tomcat 5.x
Tomcat 4.1
Tomcat 3.3
Tomcat Connectors
Archives
Documentation
Tomcat 6.0
Tomcat 5.5
Tomcat 5.0
Tomcat 4.1
Tomcat 3.3
Tomcat Connectors
Migration Guide
Tomcat 6 Downloads
Welcome to the Tomcat 6.x download page. This page provides download links for obtaining the latest versions of all Tomcat release branches, as well as links to the archives of older releases.
Quick Navigation
KEYS | 6.0.10 | Archives
Release Integrity
You must verify the integrity of the downloaded files. We provide PGP signatures for every release file. This signature should be matched against the KEYS file which contains the PGP keys of Tomcat's Release Managers. We also provide an MD5 checksum for every release file. After you download the file, you should calculate a checksum for your download, and make sure it is the same as ours.
Mirrors
Done

Figure 14-3: Downloading the KEYS file of the keys for the Tomcat release managers

Assuming you have the file locally, you can import the keys into your public key ring using this command:

```
C:\Program Files\GNU\GnuPG>gpg --import < c:\KEYS.txt
gpg: key F22C4FED: public key "Andy Armstrong <andy@tagish.com>" imported
gpg: key 86867BA6: public key "Jean-Frederic Clere (jfclere) <JFrederic.Clere@fu
```

(continued)

```
jitsu-siemens.com>" imported
gpg: key E86E29AC: public key "kevin seguin <seguin@apache.org>" imported
gpg: key 307A10A5: public key "Henri Gomez <hgomez@users.sourceforge.net>" imported
gpg: key 564C17A3: public key "Mladen Turk (*** DEFAULT SIGNING KEY ***) <mturk@
apache.org>" imported
gpg: key 7C037D42: public key "Yoav Shapira <yoavs@apache.org>" imported
gpg: key 33C60243: public key "Mark E D Thomas <markt@apache.org>" imported
gpg: key 41E49465: public key "Remy Maucherat <remm@apache.org>" imported
gpg: key 0D811BBE: public key "Yoav Shapira <yoavs@computer.org>" imported
gpg: Total number processed: 9
gpg:               imported: 9
```

With these keys in your key ring, you can then check the detached signature of the downloaded file (download from the link labeled `pgp`, right next to your Tomcat binary download) against the download. For example, if you have this signature downloaded and saved in `c:\apache-tomcat-6.0.10.zip.asc`, you can use the following command:

```
C:\Program Files\GNU\GnuPG>gpg c:\apache-tomcat-6.0.10.zip.asc
Detached signature.
Please enter name of data file: c:\downloads\apache-tomcat-6.0.10.zip
gpg: Signature made 02/13/07 09:11:31 using DSA key ID 41E49465
gpg: Good signature from "Remy Maucherat <remm@apache.org>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg: There is no indication that the signature belongs to the owner.
Primary key fingerprint: 80FF 76D8 8A96 9FE4 6108  558A 80B9 53A0 41E4 9465
```

The signature is now verified against the release manager's (Remy Maucherat) key.

> *Although the signature is verified, it is not certified by a trusted authority. At this time, depending on your security requirement, you may need 100 percent trust certification. If you want full trust certification, you need to meet the signer face to face and verify the fingerprint, or use GPG's web-of-trust features to indirectly certify it (see GPG documentation).*

Once you have ascertained the integrity of your downloads, you can have the peace of mind to deploy the binary. The next section looks at how you can secure the Tomcat server instance(s) that you install.

Securing the Tomcat Server Installation

The general strategy to use in securing a Tomcat server installation can be summed up as follows:

1. As much as possible, remove any application or Tomcat component that can be exploited as a security weakness.
2. Tie down any remaining pieces using Tomcat mechanisms and OS mechanisms.

You can see examples of this strategy applied throughout this chapter.

It is important to note that the techniques shown here are used to secure a server instance — keeping the server itself secure. Later parts of this chapter cover techniques used in securing applications that run within the server.

Removing Default Applications

By default, Tomcat ships with several Web applications installed and ready to run, including the following:

- `ROOT`: Contains the simple default welcome page
- `docs`: Tomcat documentation
- `examples`: Simple examples of JSPs and servlets demonstrating Tomcat's standards compliance
- `manager` and `host-manager`: Two powerful system applications to make administering virtual hosts and the Tomcat servers more convenient

Some of these applications present a security risk. The following sections examine the risks posed by each application and, where necessary, offer solutions to reduce or eliminate those risks.

ROOT and tomcat-docs

The `ROOT` and `tomcat-docs` applications present a very minimal risk. Many users define a new root Web application; in these situations, `ROOT` is effectively removed. Otherwise, `ROOT` should be deleted. It is implemented as a JSP so a potential exploit could exist. The risk is remote, but because the `ROOT` application provides no useful functionality, no benefit is obtained by keeping it around.

`tomcat-docs` contains no JSPs or servlets, and its content is freely available in many other locations on the Web. It poses no known security risks.

System Applications — manager and host-manager

Of the included system applications, `manager` and `host-manager` present the greatest security risks by virtue of their powerful functionality. They can potentially provide a remote access backdoor to your application and data.

If Tomcat is installed by extracting files from an archive, these two applications are effectively disabled; no account has access to use them. For maximum security, especially on production systems, system administrators should delete them completely by removing their directories:

```
$CATALINA_HOME/ webapps/host-manager
$CATALINA_HOME/webapps/manager
```

Tying Down System Application Access Security

If, for some reason, you are required to offer host-manager and/or manager functionality remotely (and therefore cannot remove the system applications), you should make sure that you choose a username and password that are difficult to guess. Changing the context and choosing a good username and password are two effective, simple ways to increase the security of the management applications. However, you can take some additional steps to lock them down even further, including the following:

- Change the application's authentication mechanism from BASIC to a more secure type. See the section "Authentication and Realms," later in this chapter for more details on this approach.
- Allow only specific client addresses (hosts) to access these applications. This is covered in the section "Host Restriction," later in this chapter.

Removing JSP and Servlet Examples

While neither of these JSP or servlet examples presents any known security risks (other than providing obvious targets for Denial of Service attacks), it's a good idea to delete them. They provide no useful functionality, and the possibility exists that attackers can exploit them. They are located in the following location:

```
$CATALINA_HOME/webapps/examples
```

Deleting the examples application is as simple as removing the `examples` directory and everything under it.

Changing the SHUTDOWN Command

By default, the Tomcat `SHUTDOWN` command works by connecting to a special Tomcat socket on port 8005 and sending the following character sequence:

```
SHUTDOWN
```

Tomcat provides no authentication mechanism to restrict clients from connecting to Tomcat, sending these characters, and shutting down Tomcat. You can try it yourself by using Telnet.

The easiest way to prevent unauthorized use of this functionality is by blocking port 8005 with a firewall. If this is not possible for whatever reason, the system administrator should change the port and the `SHUTDOWN` character sequence by editing the following line of the `TOMCAT_INSTALLATION/conf/server.xml` file:

```
<Server port="8098" shutdown="downbaby">
```

In this example, the port has been changed to `8098` and the character sequence to `downbaby`.

Running Tomcat with a Special Account

Despite the best efforts of Tomcat's authors, application developers, and system administrators, there is a chance that Tomcat can be exploited. Thus, it is prudent to consider mechanisms that prevent the amount of damage that an attacker could inflict by gaining control of Tomcat.

Perhaps the most effective damage-control mechanism is running Tomcat under its own account, an account with only those privileges necessary to run Tomcat and nothing more. If this strategy is used, hackers who gain control of Tomcat are presented with few ways to wreak havoc.

The following sections describe the process of running Tomcat with its own account.

> *Note that when running Tomcat with its own (non-root) account on Linux systems, the Tomcat server instance will not be able to bind to the privileged port 80. Binding to port 80 is desirable because users can access the server without specifying the port in the URL (for example,* `http://www.wrox.com/` *instead of* `http://www.wrox.com:8080/`*). This can be readily solved by redirecting incoming port 80 traffic of a hardware firewall/router to the Tomcat host's port 8080. Other solutions exist and are fully explored in Chapter 3.*

Creating a Non-Privileged Tomcat User

The first step in the process is to create an account for running Tomcat. For simplicity, this account is referred to as `tomcat` in the remainder of this section.

Be sure to configure the `tomcat` account with the environment variables required to run Tomcat — notably, the `JAVA_HOME` and `CATALINA_HOME` variables.

> **Restricting Account Privileges in Windows**
>
> To ensure that unintentional privileges are not extended to the account, the `tomcat` account should be removed from all groups. When creating a user from the Users and Passwords control panel (called User Accounts in Windows Vista and Windows XP), Windows automatically adds `tomcat` to at least one group. The Computer Management utility must be used to remove the `tomcat` account from all groups. System administrators may want to consider creating a special "Restricted Services" group for the `tomcat` account (more on this strategy will follow in subsequent paragraphs). Note also that the `tomcat` account should be given a password that never expires, which can be accomplished with the same Computer Management utility.

> **Restricting Account Privileges in Linux**
>
> Create both a `tomcat` user and a `tomcat` group. This is the default behavior of the `useradd` command. By assigning `tomcat` to its own new group, system administrators ensure that privileges are not unintentionally granted to the account.

Running Tomcat with the Tomcat User

After creating the `tomcat` account, the operating system must be configured to use the account when launching Tomcat.

Configuring Windows

If Tomcat is configured to run as a service (see Chapter 3 for details), you can use the Services utility to select a user account when launching Tomcat. The Services utility is in the Administrative Tools folder, which in turn is located in the Control Panel folder.

To change the account, double-click on the Tomcat service and select the Log On tab. That tab provides the capability for the service to "log on as" a specific account. Enter the Tomcat account and its password in the appropriate locations. The service should then be restarted for the new setting to take effect.

Configuring Linux

There's no one way to configure a Linux system to start Tomcat with its own user account. The basic idea is to launch Tomcat using a syntax similar to the following in whatever startup scheme is used:

```
/bin/su tomcat $CATALINA_HOME/bin/startup.sh
```

A typical configuration might use a script such as the following in `/etc/init.d` (or wherever `init` scripts are stored):

```
#!/bin/bash
RETVAL=$?
export JAVA_HOME=/usr/java/jdk160
export CATALINA_HOME=/usr/local/tomcat6
case "$1" in
 start)
        if [ -f $CATALINA_HOME/bin/startup.sh ];
          then
            echo $"Starting Tomcat"
            /bin/su tomcat $CATALINA_HOME/bin/startup.sh
        fi
        ;;
 stop)
        if [ -f $CATALINA_HOME/bin/shutdown.sh ];
          then
            echo $"Stopping Tomcat"
            /bin/su tomcat $CATALINA_HOME/bin/shutdown.sh
        fi
        ;;
 *)
        echo $"Usage: $0 {start|stop}"
        exit 1
        ;;
esac
exit $RETVAL
```

You can the configure the script in `/etc/init.d` to load on startup by creating links to it in the desired run-level directories or by using a graphical configuration client, depending on the distribution of Linux used.

Securing the File System

Configuring a Tomcat server instance to run with its own user account is useful only if the account has most of its access privileges reduced, preventing havoc from being wreaked. Effectively, this means reducing the scope of the account's file system permissions to the minimum set required to perform the job.

Windows File System

Windows has two different types of file systems: FAT32 and NTFS. FAT32 is inherently insecure and can't be "locked down." It is, however, an excellent choice for system administrators if instability, limitations, and inefficiency are considered virtuous. NTFS, conversely, has all the necessary features for restricting the `tomcat` user's capabilities.

The type of file system being used can be determined by viewing the properties of the hard drive partition in question in the My Computer window. Windows supports upgrading FAT32 partitions to NTFS. Note, however, that once a partition has been changed to NTFS, it cannot revert back to FAT32.

Access Control Lists

NTFS security is built around the concept of *access control lists (ACLs)*. Every resource in the file system (that is, files and directories) has an ACL that is associated with it. The ACL contains a list of users and

groups and the operations that the users/groups are permitted to perform. The set of allowed operations for a user or group is that entity's *permissions*.

By default, Windows allows all users to access any resource in the file system, with the exception of sensitive areas, such as the Windows directory itself and the profile resources of other users. For the purposes of securing a Tomcat installation, these permissions are too liberal.

The instructions in this section are intentionally minimal because this book is not intended for use as a Windows administration guide.

Restricting Permissions

To accomplish the stated goal of reducing `tomcat`'s permissions to the minimum required, all default permissions granted to the account must be revoked. To do this, the `tomcat` account must be explicitly denied access to every resource in the file system, and then selectively granted access to the necessary resources.

Use the following steps to revoke `tomcat`'s permissions:

1. Right-click the first drive partition in the My Computer window.
2. Select the Properties context menu item.
3. Select the Security tab.
4. Click the Add button.
5. Select the `tomcat` account.
6. Click every Deny check box.
7. Click the Advanced button.
8. Select the Reset permissions on all child objects and enable propagation of inheritable permissions check box.
9. Click OK.
10. Wait while Windows modifies every ACL in the partition's file system.
11. Repeat these steps with all partitions.

Granting Permissions

To do its job, the `tomcat` account must have permission to read and execute the JRE files. Thus, the next step in the process is to grant these permissions to the `tomcat` account. This is accomplished by a similar process to the one discussed previously. To start, select the directory containing the JRE used to run Tomcat, and view the Security properties of the directory. The `tomcat` account should be present in the list of groups and users. Removing the `tomcat` account with the Remove button is sufficient to grant access to run Tomcat. Propagating this change to all child objects is also necessary using the same process discussed previously.

For maximum security, the Everyone group should be removed from the JRE directory's ACL, and the `tomcat` user should be added to it, and given only the following permissions: Read & Execute, List Folder Contents, and Read. However, this necessitates explicitly granting these permissions to *every*

user who needs to use Java, which can become tedious. This illustrates the utility of creating a series of groups that have access to certain areas of the file system. For example, users who need access to Java can be given membership in the Java Users group. Users who need the capability to manipulate the contents of the Java directory can be given membership in a group called Java Developers. The extra time required to configure such a setup can be well worth the added security and scalability as more users are added.

The `tomcat` account also needs access to the `tomcat` directory. These permissions can be granted with the same procedure used to grant access to the `Java` directory. For maximum security, grant only read access to the following directories:

```
TOMCAT_INSTALLATION/bin
TOMCAT_INSTALLATION/lib
TOMCAT_INSTALLATION/webapps
```

Note that `TOMCAT_INSTALLATION/conf` must have write permissions to function if Tomcat's default UserDatabase implementation is used for user authentication. In addition, note that making the `TOMCAT_INSTALLATION/webapps` directory read-only can cause problems if Web applications modify files in their directories, or if the Tomcat `manager` application is used to deploy new Web applications.

Linux File System

Securing the Linux file system requires two steps: granting the `tomcat` account read and execute permissions on the JRE directory (recursively), and granting it read, write, and execute permissions on the Tomcat directory. There are numerous ways to grant these permissions. Following is one strategy:

- Recursively set the "other" permissions on the JRE directory to read and execute with the `chmod` command: `chmod -R o=rx *`
- Recursively set the owner of the Tomcat directory to the tomcat account: `chown -R tomcat: tomcat *`

For additional security, the owner, group, and other permissions for the following Tomcat directories can be set to read-only:

- `TOMCAT_INSTALLATION/bin`
- `TOMCAT_INSTALLATION/lib`
- `TOMCAT_INSTALLATION/webapps`

Note that `TOMCAT_INSTALLATION/conf` must have write permissions to function if Tomcat's default UserDatabase implementation is used for user authentication. In addition, note that making the `TOMCAT_INSTALLATION /webapps` directory read-only can cause problems if Web applications modify files in their directories, or if the Tomcat `manager` application is used to deploy new Web applications.

Securing the Java Virtual Machine

Configuring the file system for maximum security is an important part of securing an installation of a Tomcat server instance. Java's Security Manager architecture exposes an entirely different level of configurability. With the Security Manager, Java applications can be restricted from accessing features of the Java language and platform in a remarkably fine-grained manner.

This security architecture is turned off by default, but it can be turned on at any time. In this section, we review the Security Manager architecture in general terms. This is followed by a discussion of how this architecture specifically applies to Tomcat.

Overview of the Security Manager

As with the file system, the Security Manager architecture is based on the concept of permissions. Once the Security Manager is turned on (using a command-line switch that will be discussed shortly), applications must have *explicit permission* to perform certain security-sensitive tasks (such as creating a custom class loader or opening a network socket).

To make effective use of the Security Manager architecture, it is therefore necessary to know how to grant permissions to applications and to understand the set of possible permissions.

Granting Permissions to Applications

Policy files are the mechanism used by the Security Manager to grant permissions to applications. Policy files are simple text files composed of individual actions that applications are allowed to perform.

> *Note that the term "applications" in this context refers to applications running in the Java VM, of which the Tomcat server is a member. This is not referring to the Web applications that run within a Tomcat server instance.*

A policy file is composed of `grant` entries, which look like the following:

```
// first grant entry
grant {
  permission java.lang.RuntimePermission "stopThread";
}
// second grant entry
grant codeBase "file:${java.home}/lib/ext/*" {
  permission java.security.AllPermission;
};
```

The first `grant` entry in this example demonstrates the simplicity of the syntax. It grants all applications the capability to access the deprecated `Thread.stop()` method.

The second `grant` entry illustrates that code in specific locations can also be granted permissions. This is, of course, useful for extending permissions to certain trusted code while denying permissions to all other code. In this case, all code in the `$JAVA_HOME/lib/ext` directory is granted all permissions, which effectively disables the Security Manager architecture for that code.

Grant Entry Syntax

Each `grant` entry must be composed of the following syntax:

```
grant codeBase "URL" {
  // this is a comment
  permission permission_class_name "target_name", "action";
  ...
};
```

Note that comments in policy files must begin with `//` on each line. As shown in the first `grant` entry earlier, the `codeBase` attribute is optional. `codeBase` specifies a URL to which all the permissions should apply. The syntax is shown in the following table.

codeBase Example	Description
`file:/C:/myapp/`	Indicates that code in the directory `c:\myapp` will be assigned the permissions in the `grant` block. Note that the slash (`/`) indicates that only class files in the directory will receive the permissions, not any JAR files or subdirectories.
`http://java.sun.com/*`	All code from the specified URL will be granted the permissions. In this case, the "`/*`" at the end of the URL indicates that all class files and JAR files will be assigned the permissions, but not any subdirectories.
`file:/funstuff/-`	All code in the `/funstuff` directory will be granted the permissions. The slash (`/-`) indicates that all class files and JAR files in the directory and its subdirectories will be assigned the permissions.

Within the `grant` block, one or more permissions can be assigned. Each permission consists of a permission class name and, in some cases, an additional *target* that identifies a specific permission within the permission class. Some permission targets can additionally take parameters, called *actions*. Following are some examples of permissions:

```
grant {
  // allows applications to listen on all ports
  permission java.net.SocketPermission "localhost", "listen";
  // allows applications to read the "java.version" property
  permission java.util.PropertyPermission "java.version", "read";
}
```

Available Permissions

Permissions are defined by special classes that ultimately inherit from the abstract class `java.security.Permission`. Most permission classes define special targets that represent a security permission that can be turned on and off.

For example, the `java.lang.RuntimePermission` class defines the targets shown in the following table. (Note that this is not a complete list.)

Target Name	Description
`createClassLoader`	Allows an application to create a custom class loader
`exitVM.{n}`	Allows an application to exit the JVM via the `System.exit(n)` method

As of Java SE 6, there are 19 different permission classes offering control over various permissions. The following table shows a partial list of these classes to demonstrate the breadth of what is possible with permissions. This list is not an exhaustive listing of all possible permission targets. All of the permissions

are also available with Java SE 5. The complete list of permission classes and their targets can be viewed at the following URL:

```
http://java.sun.com/javase/6/docs/technotes/guides/security/permissions.html:
```

Permission Class	Description
`java.security.AllPermission`	By granting this permission, all other permissions are also granted. Granting this permission is the same as disabling the Security Manager for the affected code.
`java.security.SecurityPermission`	Allows programmatic access to various security features of the Java language.
`java.security.UnresolvedPermission`	This permission class is not defined in policy files. Rather, it is used as a placeholder when a policy file makes reference to a user-defined permission class that had not been loaded at the time of processing the policy file. This permission is relevant only to those interacting with the Security Manager system programmatically at runtime.
`java.awt.AWTPermission`	Controls various AWT permissions
`java.io.FilePermission`	Restricts read, write, execute, and delete access to files in specified paths.
`java.io.SerializablePermission`	Allows serialization permissions.
`java.lang.reflect.ReflectPermission`	Allows applications to circumvent the `public` and `private` mechanism's access checks and reflectively access any method.
`java.lang.RuntimePermission`	Allows access to key runtime features (such as creating class loaders, exiting the VM, and reassigning `STDIN`, `STDOUT`, and `STDERR`).
`java.net.NetPermission`	Allows various network permissions.
`java.net.SocketPermission`	Allows incoming socket connections, outgoing connections, listening on ports, and resolving host names. These permissions can be defined for specific host names and port combinations.
`java.sql.SQLPermission`	While this may sound intriguing, it controls only a single permission: setting the JDBC log output writer. This file is considered sensitive because it may contain usernames and passwords.
`java.util.PropertyPermission`	Controls whether properties can be read from or written to.
`java.util.logging.LoggingPermission`	Allows the capability to configure the logging system.
`javax.net.ssl.SSLPermission`	Allows the capability to access SSL-related network functionality.

Table continued on following page

Permission Class	Description
`javax.security.auth.AuthPermission`	Controls authentication permissions.
`javax.security.auth.` and `PrivateCredentialPermission`	Controls various security permissions.
`javax.security.auth.kerberos.` and `DelegationPermission`	Controls various security permissions related to the Kerberos protocol.
`javax.security.auth.kerberos.` and `ServicePermission`	Controls various security permissions related to the Kerberos protocol.
`javax.sound.sampled.AudioPermission`	Controls access to the sound system.

Enabling the Security Manager System

The Security Manager system is enabled by passing the `-Djava.security.manager` parameter to the Java Virtual Machine at startup, in the following manner:

```
$ java -Djava.security.manager MyClass
```

By default, Java looks for the file `$JAVA_HOME/lib/security/java.policy` to determine what permissions to grant when the Security Manager is turned on.

For more information on enabling the Security Manager and using custom policy files, see the following URL:

```
http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html
```

Advanced Security Manager Topics

Some additional Security Manager topics are simply beyond the scope of this chapter. For example, it is possible to subclass the default Java Security Manager implementation to provide for custom permission classes. It is further possible to define `grant` blocks in policy files based on code signatures. For information on these and other advanced topics, check out the following URL:

```
http://java.sun.com/javase/6/docs/technotes/guides/security/
```

Using the Security Manager with Tomcat

Now that we've covered the basics of the Security Manager system, we'll explain their use with Tomcat.

Enabling Tomcat's Security Manager

The preferred way to start Tomcat with the Security Manager enabled on Linux systems is as follows:

```
$ TOMCAT_INSTALLATION/bin/catalina.sh start -security
```

On Windows systems, the command is quite similar:

```
> TOMCAT_INSTALLATION\bin\catalina start -security
```

Tomcat's Policy File

Tomcat uses the `$CATALINA_HOME/conf/catalina.policy` file to determine its own permissions and those of its Web applications.

What follows is the file as of Tomcat 6. Note that it is divided into three sections: system code permissions, Catalina code permissions, and Web application code permissions.

System Code Permissions

Tomcat's policy file grants all permissions to the `javac` tool, which is used to compile JSPs into servlets, and grants all permissions to any Java standard extensions. Four `grant` lines are used instead of two to deal with multiple path possibilities. Note that administrators may need to add additional `grant`s to this section if the JRE used to run Tomcat uses different paths for its standard extensions (such as Mac OS X) and Tomcat Web applications are using JARs or classes in those paths.

```
// ========== SYSTEM CODE PERMISSIONS =========================================
// These permissions apply to javac
grant codeBase "file:${java.home}/lib/-" {
        permission java.security.AllPermission;
};
// These permissions apply to all shared system extensions
grant codeBase "file:${java.home}/jre/lib/ext/-" {
        permission java.security.AllPermission;
};
// These permissions apply to javac when ${java.home] points at $JAVA_HOME/jre
grant codeBase "file:${java.home}/../lib/-" {
        permission java.security.AllPermission;
};
// These permissions apply to all shared system extensions when
// ${java.home} points at $JAVA_HOME/jre
grant codeBase "file:${java.home}/lib/ext/-" {
        permission java.security.AllPermission;
};
```

Catalina Code Permissions

Note that Catalina grants all permissions to the following:

- Tomcat's startup classes (`${catalina.home}/bin/commons-daemon.jar` and `${catalina.home}/bin/bootstrap.jar`)
- Classes from the logging API (`${catalina.home}/bin/tomcat-juli.jar`)
- Shared files between class loaders (`$CATALINA/lib`)

```
// These permissions apply to the daemon code
grant codeBase "file:${catalina.home}/bin/commons-daemon.jar" {
        permission java.security.AllPermission;
};
```

(continued)

```
// These permissions apply to the logging API
grant codeBase "file:${catalina.home}/bin/tomcat-juli.jar" {
        permission java.security.AllPermission;
};
// These permissions apply to the server startup code
grant codeBase "file:${catalina.home}/bin/bootstrap.jar" {
        permission java.security.AllPermission;
};
// These permissions apply to the servlet API classes
// and those that are shared across all class loaders
// located in the "lib" directory
grant codeBase "file:${catalina.home}/lib/-" {
        permission java.security.AllPermission;
};
```

System Properties Access Permissions

Tomcat grants read-only access (to Web applications running within the Tomcat server instance) to various system properties and other miscellaneous permissions as commented here:

```
// ========== WEB APPLICATION PERMISSIONS =====================================
// These permissions are granted by default to all web applications
// In addition, a web application will be given a read FilePermission
// and JndiPermission for all files and directories in its document root.
grant {
    // Required for JNDI lookup of named JDBC DataSource's and
    // javamail named MimePart DataSource used to send mail
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "java.naming.*", "read";
    permission java.util.PropertyPermission "javax.sql.*", "read";
    // OS Specific properties to allow read access
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
    permission java.util.PropertyPermission "os.arch", "read";
    permission java.util.PropertyPermission "file.separator", "read";
    permission java.util.PropertyPermission "path.separator", "read";
    permission java.util.PropertyPermission "line.separator", "read";
    // JVM properties to allow read access
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission "java.vendor.url", "read";
    permission java.util.PropertyPermission "java.class.version", "read";
 permission java.util.PropertyPermission "java.specification.version", "read";
 permission java.util.PropertyPermission "java.specification.vendor", "read";
 permission java.util.PropertyPermission "java.specification.name", "read";
 permission java.util.PropertyPermission "java.vm.specification.version", "read";
 permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
 permission java.util.PropertyPermission "java.vm.specification.name", "read";
 permission java.util.PropertyPermission "java.vm.version", "read";
 permission java.util.PropertyPermission "java.vm.vendor", "read";
 permission java.util.PropertyPermission "java.vm.name", "read";
    // Required for OpenJMX
    permission java.lang.RuntimePermission "getAttribute";
 // Allow read of JAXP compliant XML parser debug
 permission java.util.PropertyPermission "jaxp.debug", "read";
```

```
    // Precompiled JSPs need access to this package.
    permission java.lang.RuntimePermission "accessClassInPackage.org.apache.jasper
.runtime";
    permission java.lang.RuntimePermission "accessClassInPackage.org.apache.jasper
.runtime.*";
};
```

Note that system administrators are not only free to modify Tomcat's policy file, they are encouraged to do so. Once the Security Manager has been enabled, it's likely that changes to it will be required in order for certain aspects of deployed Web applications to function.

Recommended Security Manager Practices

You have explored the process of enabling the Security Manager with Tomcat, and are familiar with the location of Tomcat's policy file. The following are recommended practices for granting permissions to applications.

These techniques prevent tampering of the underlying operating system, other machines on the network, and the Tomcat server instance, from potentially malicious Web applications.

Using the Security Manager

If the Security Manager is not used with Tomcat, any JSP or class file in a badly written or malicious Web application is free to perform any action on the server machine that it desires. This includes opening unauthorized connections to other network hosts, writing to the server file system where is shouldn't, or abnormally terminating Tomcat itself by issuing the `System.exit(n)` command.

Clearly, to maintain a secure Tomcat installation, the Security Manager should be enabled, and fine-grained permissions should be set.

Understanding Application Requirements

If Tomcat's default policy file is enabled, Web applications are likely to be unable to perform certain required functions. Consider the following tasks that are unauthorized with Tomcat's default policy configuration:

- Creating a class loader
- Accessing a database via a socket (for example, the MySQL JDBC driver trying to establish a connection with a MySQL database)
- Sending an e-mail via the JavaMail API
- Reading or writing to files outside of the Web application's directory

There are a myriad of permissions that an application may require. System administrators must communicate with the application developers to understand which permissions the Web applications will require.

Examples for enabling some of the common permissions listed here are reviewed in the next section. To learn about other permissions, review the Java Security documentation links provided earlier in this chapter.

Enabling Creation of a Class Loader

The following example shows how to give a specific Web application, `yourWebApp`, the capability to create a class loader:

```
grant codeBase "file:${catalina.home}/webapps/yourWebApp/WEB-INF/classes/-" {
  permission java.lang.RuntimePermission "createClassLoader";
};
```

Enabling JDBC Drivers to Open Socket Connections to Databases

The following example shows how to allow all Web applications access to a specific database running on the host `db.server.com` on port 54321:

```
grant codeBase "file:${catalina.home}/webapps/-" {
  permission java.net.SocketPermission "db.server.com:54321", "connect";
};
```

Note that the preceding example allows all code across all of your Web applications to connect to `db.server.com:54321`. If this is too much of a security risk, the JDBC driver can be explicitly granted permission individually:

```
grant codeBase "file:${catalina.home}/webapps/webAppName/WEB-INF/lib/
mysql-connector-java-5.0.4-bin.jar" {
  permission java.net.SocketPermission "db.server.com:54321", "connect";
};
```

Sending E-Mail with JavaMail

Sending e-mail requires that a Web application have access to port 25 on an SMTP server. The following example shows how to grant this permission to all classes in a Web application:

```
grant codeBase "file:${catalina.home}/webapps/myWebApp/WEB-INF/classes/-" {
  permission java.net.SocketPermission "mail.server.com:25", "connect";
};
```

Reading or Writing to Files Outside of the Web Application's Directory

Earlier in this chapter, we discussed securing the file system. If the file system has been properly secured, the following grant can be used to give Web applications full access to the file system (and thus rely on the operating system to enforce permissions):

```
grant {
  java.io.FilePermission "<<ALL FILES>>", "read,write,execute,delete";
};
```

While it may be tempting to use the Java Security Model in place of securing the file system via operating system permissions, such a tactic is unwise. Relying on the operating system provides an important extra layer of security in the event that the Java Virtual Machine itself becomes compromised and exploited. Additionally, in many configurations, it is likely that Tomcat is not the only exploitable network service on the server — another good reason to utilize the operating system's security model, as Tomcat's security settings would not apply to the other services.

In addition, by default, all Java applications do have read access to the directory in which they are located, including its subdirectories.

Securing Web Applications

The previous sections have been concerned with securing the Tomcat server instance, the underlying operating system resources, and the Java platform. Up until this point, you have not examined any techniques to secure specific Web applications running in the Tomcat instance. In this section, we consider techniques for securing individual Web applications. These techniques fall under the following categories:

- Authentication and Realms
- Encryption
- Host Restriction

Note that these techniques can also be applied to Tomcat's built-in system applications.

The techniques described here are often called declarative security measures. They are declarative because you apply them by modifying XML configuration files, typically the application's `web.xml` file, and apply them declaratively. This is in contrast to programmatic security, in which Java coding is used within a Web application to perform security checks. Both of these terms are official Java EE terminology and concepts. As an example of programmatic security, a sensitive Web application can check, using Java coding, to make sure that any user accessing itself must have the role of manager (or higher) before allowing access. An in-depth discussion of declarative security is beyond the scope of this book, but any Java EE 5 book should have in-depth exploration into programmatic security.

> *Note that many applications designed for running in Tomcat do not enforce programmatic security, and do not use declarative security in their* `web.xml` *file. These applications, for the most part, are vulnerable. You can add a minimal level of security by enforcing user authentication via the techniques shown in this section.*

Authentication and Realms

Authentication is the process of determining and validating the identity of an application's client. The Servlet specification provides integration with the Java Authentication and Authorization Service (JAAS) API. This enables Web applications to authenticate their users in a standard way that is portable across different Servlet containers.

> *Some Java developers have been known to eschew open standards in favor of their own. It is entirely possible (and indeed somewhat common) for Servlet developers to authenticate users via some home-grown mechanism, rather than via the JAAS/Servlet standard mechanism discussed subsequently in this section. System administrators should be aware that in such circumstances, this section will be of little utility.*

Tomcat provides a Realm mechanism, mandated in the Servlet specification, to assist Web applications in the implementation of user authentication. Essentially, Realms hold authentication information that can be accessed via either programmatic security, or via declarative security (configuration files). Details of the available Realms are explored later in this chapter.

Authentication Mechanisms

Servlet-based applications have four standards-based authentication mechanisms from which to choose:

- BASIC
- DIGEST
- Form
- HTTPS Client Certificate

A brief description of these mechanisms follows. We demonstrate their use later in the chapter.

BASIC

As its name implies, the BASIC authentication mechanism is simplistic. The browser sends base64-encoded credentials to the server, which then decodes them and uses them to authenticate the user. This mechanism has two somewhat serious problems:

- Base64 encoding is not secure. Base64 is intended as a means of encoding binary data as ASCII data for transmission via protocols that lack support for binary data. It is not a type of secure encryption mechanism. In the case of the BASIC authentication mechanism, base64 is better than sending credentials in plaintext, but not much better.
- Browsers cache credentials after authentication. Once a user authenticates, there is no way for the user to log out other than by exiting the browser or if the server times out the session because of inactivity. This disadvantage also applies to the other browser-managed authentication mechanisms, such as DIGEST and HTTPS Client Certificate.

Nevertheless, despite its insecurity, BASIC remains a good option for a simple level of security designed to keep out the "mindless hordes." When administrators really don't care if the protected resource is compromised, BASIC is not a bad mechanism to use.

DIGEST

DIGEST is a step up from BASIC. Another browser-based mechanism, DIGEST is very similar to BASIC with the exception that the password is transmitted in a secure fashion. The browser performs a *digest* on the password (a digest is a one-way hash, as explained shortly) and transmits the digest to the server. The server then digests the password to which the browser-provided password digest will be compared, and if the two match, the authentication is successful.

DIGEST is reasonably secure, but it too suffers from two flaws:

- In Tomcat, the original password must be stored somewhere in plaintext. This is especially unfortunate when the password is stored in a file, as it can then be viewed by anyone with access to that file on the server machine. (A workaround is possible using file permissions to secure access to the file.)
- It has the same cached credential problem that BASIC has. (See the preceding section, "BASIC," for details.)

A digest, also called a *hash*, is used to provide proof that a set of data hasn't been nefariously (or unintentionally) altered.

A hashing algorithm takes some data as input and from it creates a unique fingerprint (which is usually 16 or 20 bytes long). This is a one-way process, meaning that the digest cannot be undigested to discover the original data. Because each fingerprint is unique, the digest of the original data can be compared with a digest of a second set of data. If the digests match, then the second set of data is proved to be identical to the original digest of data. If two sets of data are purported to be identical, they are confirmed as such.

This process can be applied to passwords by digesting the password and storing its digest in a file or database. Thus, even if the stored password digest is compromised, an attacker cannot "undigest" the password the hash represents, and it is thus unusable. To determine whether a user has entered the same password, the user's password is digested and compared with the digest value on file. If they match, it is the same password.

Java supports two digest algorithms:

- **MD5:** This algorithm is used in several password-storage mechanisms, including many Unix systems. MD5 produces a 16-byte (128 bits) message digest.
- **SHA:** This algorithm is more secure than MD5 and produces a 20-byte message digest.

Form

In form-based authentication, the browser does not knowingly cooperate in the authentication process. Instead, the Web application creates an HTML form wherein the form name and username and password fields all have special names. These fields can then be intercepted by the Servlet container, which uses the data to provide authentication.

Because an HTML form can be transmitted over an encrypted connection (HTTPS), form-based authentication can be made reasonably secure. It does suffer from at least one disadvantage, however:

- Reliance on usernames/passwords as credentials. While the form-based mechanism can transmit credentials after they have been encrypted with HTTPS, the authentication mechanism is still reliant on passwords, which can be defeated either by brute force or by social engineering.

HTTPS Client Certificate

When a browser establishes a secure connection with a server, the browser is sent a public key certificate from the server. This certificate enables the browser to authenticate the server. That is, it enables the browser to know the true identity of the server as certified (signed) by a trusted third party (such as VeriSign). This authentication mechanism enables the browser to be certain of the identity of the server, so that sensitive transactions such as e-commerce can be conducted. Note, however, that this process is asymmetric; the server does not receive a certificate from the client.

The HTTPS client certificate mechanism upgrades this process to be symmetrical. With this mechanism, the Web browser transmits a public key certificate to the server, which can then use the certificate to authenticate the client. Both parties, therefore, are authenticated with each other. Note, however, that most server-based applications rely on simpler mechanisms to authenticate their clients (such as an HTML form-based mechanism).

The HTTPS client certificate mechanism is, of course, quite secure. If the public key architecture upon which HTTPS client authentication is based were defeated, the very basis of secure e-commerce would fall with it. Beyond this apocalyptic scenario, however, are some potential weaknesses:

- **Key length:** The most important factor in the security of public key encryption is the length of the key used to encrypt the messages. As computing evolves and computing power increases, ever larger keys will be needed to maintain security against brute force hack attempts. Administrators should stay informed about public key architecture issues and upgrade the keys used should this become necessary in the future.
- **Theft:** The fundamental assumption of public key authentication is that the corresponding private key is available only to the trusted party. Should the private key be stolen, the authentication would be compromised.

While quite secure, the HTTPS client certificate mechanism is rarely used outside of business-to-business applications because of the complexity of the process one must go through (and associated cost) to obtain a certificate for each authenticating client.

Configuring Authentication

In order for a Web application to use one of the authentication mechanisms just described, it must be configured to do so inside its deployment descriptor (`web.xml` file). This is accomplished by adding `<security-constraint>` and `<login-config>` elements to the `<web-app>` element. These elements are discussed in Chapter 7. An example of their use is shown here:

```
<web-app ...>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Entire Application</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
       <role-name>manager</role-name>
    </auth-constraint>
  </security-constraint>
  <login-config>
    <auth-method>FORM</auth-method>
    <realm-name>My Application</realm-name>
    <form-login-config>
      <form-login-page>/login.jsp</form-login-page>
      <form-error-page>/notAuthenticated.jsp</form-error-page>
    </form-login-config>
  </login-config>
  ...
  <security-role>
      <role-name>manager</role-name>
  </security-role>
</web-app>
```

In this code excerpt, the `<security-constraint>` element is used to define a portion of the application that is restricted to users belonging to a specific role. The `<url-pattern>` element uses URL pattern matching to determine the protected portion of the application (in this case, the entire application), and the `<role-name>` element is used to restrict that portion of the application to authenticated users who belong to the "user" role. For more information on roles, see the section "Users and Roles," later in this chapter.

The <login-config> element is used to specify how users authenticate with the Web application. <auth-method> determines which of the authentication mechanisms described here is used. Possible values include BASIC, DIGEST, FORM, and CLIENT-CERT. Because we've chosen FORM, the <form-login-config> element must be nested in the <login-config> element. <form-login-config> identifies which page in the Web application is used to authenticate the user (/login.jsp) and which page is displayed when authentication fails (/notAuthenticated.jsp). No page is configured to be displayed when authentication succeeds. Instead, the user is presented with the URL that triggered the authentication in the first place.

Authentication Form

In the preceding example, the URL /login.jsp is used to specify the login form. While any valid HTML page containing an HTML form may be used, the HTML form used to send the credentials to the server must be configured in three specific ways:

- The value of its <form> element's action attribute must be j_security_check.
- The username must be sent in a field named j_username.
- The password must be sent in a field named j_password.

For this example, create a Web application named secure. In a Web application directory named "secure," place the following login.jsp into it.

Following is an example of a conforming form:

```
<html>
  <head><title>Please Log In</title>
  <body>
    <form method="POST" action="<%= response.encodeURL("j_security_check") %>">
      <table>
        <tr>
          <th>Username:</th>
          <td><input type="text" name="j_username"></td>
        </tr>
        <tr>
          <th>Password:</th>
          <td><input type="password" name="j_password"></td>
        </tr>
        <tr>
          <td><input type="submit" value="Log In"></td>
          <td><input type="reset"></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

The error page, named notAuthenticated.jsp, can contain any HTML that conveys to the user the fact that the authentication attempt failed.

You should also create a page named index.jsp that contains the fictitious main page of the application. This page is shown if your authentication is successful.

Security Realms

The authentication mechanism descriptions detailed how the credentials used for the authentication process (for example, username and password) are obtained. However, for authentication to take place, Tomcat must also have access to the real credentials against which those sent from the browser must be compared. This section describes where Tomcat stores the actual credentials on the server and how it obtains them.

Realms are the standard mechanism used for storing the credentials used by Tomcat to authenticate the client. Tomcat's Realm mechanism is an implementation of the Realm support mandated in the Servlet specification.

A Realm is a standard programming interface defined in Tomcat for accessing a user's username, password, and roles. Tomcat 6's built-in default authentication implementations (including the login mechanisms for the `manager` utility and the Single Sign-on Valve) depend on Realms to authenticate the user.

Users and Roles

The Web application security model is built around the concept of users and roles. Users are assigned to a role, which determines the resources that the user is allowed to access. For example, a Web application can declare that the resource `/admin` can be accessed only by users belonging to the "admin" role. Then, a Realm can be configured to consider the users "alice" and "bob" as belonging to the "admin" role. Thus, when "alice" and "bob" authenticate, they will be allowed access to `/admin`.

The advantage of roles is that they enable the Web application to be configured independently of the permissions of the users who access the application. Using the preceding example, the deployment descriptor of the application needs to specify only that a "manager" role is required, and is not concerned with the identities of the users who are allowed access.

The actual mapping of users to roles can be specified at deployment time — and can be changed dynamically without having to change the application code. This clean separation of the authentication code from the actual method of authentication is the main advantage of Realms. This separation allows for many different ways of creating Realms. The following four built-in Realm implementations can be deployed with Tomcat 6:

- File-backed, in-memory Realms
- JDBC Realms
- JNDI-based Realms
- JAAS-based Realms

In addition to these built-in Realms, it is also possible for developers to create custom Realms — supplying the authentication data via arbitrary custom means.

The following sections provide detailed coverage of each of the built-in Tomcat 6 Realm implementations. Where applicable, a basic deployment configuration is first described to familiarize you with the particular Realm implementation, followed by the presentation of a more secured method of deployment.

File-Based Realm: UserDatabase

A file-based Realm maintains its authentication data in flat files. These files can be edited using a normal text editor. The data is kept in human-readable format (such as XML). The primary built-in file-based Realm implementation for Tomcat 6 is called *UserDatabase*.

UserDatabase reads authentication data from a specified XML file for use by Tomcat 6 during startup. This realm also has the following properties:

- The data in the Realm can be programmatically changed during the lifetime of the engine. This enables various possibilities for building administrative utilities.
- UserDatabase is persistent. That is, upon modification and shutdown, the UserDatabase can also persist any changes back to its associated XML (`tomcat-users.xml`) data file.
- The `admin` (not yet available with TC6) utility supports the graphical editing of authentication data within a UserDatabase Realm.

The UserDatabase realm is an integral part of Tomcat 6's authentication and programmatic security support.

Configuring UserDatabase

In the default `server.xml` (in the Tomcat 6 server distribution), the UserDatabase Realm is already configured. The UserDatabase is typically configured in the `<GlobalNamingResources>` element as a JNDI Resource. Here is a typical configuration:

```
<Resource name="UserDatabase" auth="Container"
              type="org.apache.catalina.UserDatabase"
              description="User database that can be updated and saved"
              factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
              pathname="conf/tomcat-users.xml" />
```

This makes the UserDatabase accessible from an application via JNDI lookup, relative to the `java:comp/env` naming context. Furthermore, it also provides an easy reference in a later scope. For example, you can use the UserDatabase as a Realm at the `<Engine>` container level by adding the following `<Realm>` definition:

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                 resourceName="UserDatabase"/>
```

In fact, this is precisely the content of the default Tomcat 6 `server.xml` file. This means that both the `manager` application and the `host-manager` system applications actually rely on UserDatabase as the Realm for authentication.

To see how UserDatabase is a modifiable, updateable Realm, use a text editor to add a new user/password entry:

1. Find the `$CATALINA_HOME/conf/tomcat-users.xml` file and add the following entry using the text editor:

```
<user username="joe" password="joe" fullName="Joe Smoe" roles="manager,role1"/>
```

2. Close your browser and start a new instance, and then try to access the `manager` system application using the new user `joe`. The UserDatabase has been updated, and the authentication succeeds, without the need to stop and start the Tomcat server.

In the approach detailed here, the username and password used for authentication are stored on the server in plaintext. The next section describes how to secure a file-based Realm.

Securing a File-Based UserDatabase Realm

A UserDatabase Realm can be configured in a more secure manner than previously illustrated. While UserDatabase can be made reasonably secure, the ideal solution for secure authentication is to use an alternative Realm (JDBC, JNDI, or JAAS), which is discussed shortly.

The UserDatabase Realm stores passwords in cleartext in the `tomcat-users.xml` file. This is not very secure. Therefore, a way must be found to store these passwords in a less readable format. Use the following steps to configure UserDatabase in a secure fashion:

1. Select the password digest algorithm.
2. Create a digested password.
3. Add the digested password to the Realm.
4. Test the digested password.

Selecting the DIGEST Algorithm

The choice of a digest algorithm is limited to those supported by the `java.security.MessageDigest` class (typically SHA or MD5). To choose one, the `digest` attribute of the `<Realm>` element in the `$CATALINA/conf/server.xml` file must be set. In this example, SHA is used:

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
resourceName="UserDatabase" digest="sha" />
```

When a user enters a password at the authentication stage, Tomcat digests it with the algorithm specified here and then compares it with the value stored in the authentication file.

Creating a DIGESTed Password

A digested version of the password must now be created. Tomcat comes with a script (`digest.sh` on Linux; `digest.bat` on Windows) located in `$CATALINA/bin` that calculates digests. The algorithm to use (SHA in this case) and the string to digest (`tomcat`, which is our password) must be specified as parameters:

```
$ $CATALINA_HOME/bin/digest -a sha tomcat
tomcat:536c0b339345616c1b33caf454454d8b8a190d6c
```

The output (highlighted in bold) is the string entered, followed by a colon and the SHA hash needed.

Adding the DIGESTed Password to the UserDatabase Realm

The final step is to add the digested password to the UserDatabase Realm for the Tomcat installation. This is accomplished by copying the digested output of the preceding step and adding it as the `password` attribute of a user in `tomcat-users.xml`:

```
<?xml version="1.0"?>
<tomcat-users>
  <role rolename="manager"/>
  <user username="maharaja"
        password="536c0b339345616c1b33caf454454d8b8a190d6c"
        roles="manager"/>
</tomcat-users>
```

Here, a user named `maharaja` with the role of `manager` has been added. This role can access the `manager` application that ships with Tomcat, as well as the example Web application in the download for this chapter.

Testing the DIGESTed Password

The digested password can be tested by accessing the example Web application. Browse to the following URL:

```
http://localhost:8080/secure/index.jsp
```

A login page should be presented. Enter **maharaja** as the User name and **tomcat** as the password and click the Log In button. If all goes well, access to the application is granted, and you see the `index.jsp` file rendered.

File-based Realms (such as UserDatabase) are easy to configure and do not depend on external resources to operate. However, they are rather limited because all authentication and authorization data must reside in a file. When the size of the data is large, file-based Realms can become inefficient to manage. The security of file-based Realms is also rather limited. By using an external relational database for authentication data, JDBC-based Realms overcome these limitations. The next section explores the administration of JDBC Realms.

JDBC Realms

A JDBC Realm is a Realm implementation that uses tables maintained in a relational database (such as MySQL or Oracle). Authentication and authorization data reside in an external database, potentially an existing one containing user data. Unlike file-based Realms, JDBC Realms enable the flexible addition, updating, modification, and deletion of authentication data and user/role mappings. Because data in an RDBMS is maintained dynamically, any changes that are made to the content of the authentication data are immediately reflected in the Realm. In addition to these advantages, sophisticated maintenance and administration tools can be readily created using JDBC to access and maintain the tables within the Realm.

Mapping Columns to the Required View

The JDBC Realm implementation in Tomcat 6 has a particular view of how the tables in the Realm must be maintained. Fortunately, the configurable parameters of Realms enable you to map to any existing schema containing the same data.

More specifically, the JDBC Realm implementation expects the following tables — in a standard normalized relation.

Table Name	Description
`users`	Contains username and password information
`user_roles`	Contains user-to-roles mapping information

The `users` table is expected to contain the following two columns as a minimum. It has `login` as the primary key (indexed).

Column	Type	Length
`login`	`varchar` not `null`	15
`password`	`varchar` not `null`	15

The `user_roles` table is expected to contain the following two columns as a minimum.

Column	Type	Length
`Login`	`varchar` not `null`	15
`role`	`varchar` not `null`	15

Note that the datatype can be any type that results in a character string, and longer length fields will be accommodated.

The compound primary key in this table is `{login, role}`. This means that a single user can have multiple roles. The `login` column in both the `users` and `user_roles` tables can be relationally joined during regular queries.

For maximum flexibility, the mentioned table names and column names are not imposed on the underlying table. Instead, they are mapped during Tomcat runtime to the underlying table. The mapping is specified in the configuration of the `Realm` element.

The JDBC Realm implementation, contained in the `org.apache.catalina.realm.JDBCRealm` class, will assume this configuration while using a JDBC driver to access the data in the Realm.

Realm definitions must be configured in a Realm element within the scope of any container component. Specifically, the JDBC Realm implementation may be configured with the attributes shown in the following table.

Attribute	Description	Required?
`className`	The Java programming language class that implements the JDBC Realm. This should be the implementation provided by Tomcat — `org.apache.catalina.realm.JDBCRealm`.	Yes
`connectionName`	The JDBC connection username to be used.	Yes

Attribute	Description	Required?
connectionPassword	The JDBC connection password to be used.	Yes
connectionURL	The JDBC connection URL used to access the database instance.	Yes
digest	Specifies the digest algorithm used when the Container Managed Security uses the digest method of authentication. Takes a value that specifies the digest algorithm, such as SHA, MD2, MD5, and so on. (For a complete list of current values, consult the Javadoc java.security.MessageDigest class.)	No
driverName	Name of the JDBC driver, a Java programming language class name.	Yes
userTable	The actual name of the table in the database that matches the Users table in the required view.	Yes
userNameCol	The actual column name of the column in both the userTable and userRoleTable that matches the user column in the required view.	Yes
userCredCol	The name of the column in the userTable that matches the password column in the required view.	Yes
userRoleTable	The actual name of the table in the database that matches the user_roles table in the required view.	Yes
roleNameCol	The name of the column in the userRoleTable that matches the role_name column in the required view.	Yes

The combination of the attributes userTable, userNameCol, userCredCol, userRoleTable, and roleNameCol enables you to map the existing database table and columns containing authentication and role information to the view required by the Realm.

Configuring JDBC Realms with Digested Passwords

To gain some experience in configuring JDBC Realms, an external MySQL database server acts as the example relational database system. MySQL is free for download and is available for Linux and Windows; download a copy at www.mysql.com. The installation of MySQL is not covered in this chapter.

Setting up MySQL Tables

For JDBC authentication, Tomcat requires a database with at least two tables: users and user_roles. The database is named authority. Here's the SQL to create the database:

```
CREATE DATABASE IF NOT EXISTS authority;
USE authority;
CREATE TABLE users (
   login VARCHAR(15) NOT NULL PRIMARY KEY,
   password VARCHAR(32) NOT NULL
);
```

(continued)

```
CREATE TABLE user_roles (
   login VARCHAR(15) NOT NULL,
   role VARCHAR(10) NOT NULL,
   PRIMARY KEY (login, role)
);
```

This code can be entered into MySQL interactively through its console, or run as a script:

```
$ mysql < authority.sql
```

After creating the database and tables, users and roles must be added. To make the installation secure, the passwords of the users stored in the `authority` table will be digested. Most databases provide functions for digesting information, and MySQL is no exception. The `MD5()` function will be used in this case. Here's the SQL to add an `admin` user:

```
INSERT INTO users (login, password) VALUES ('maharaja', MD5('tomcat'));
```

The JDBC Realm has an attribute called `digest` that is used to specify the digest algorithm to use on the password entered at the authentication stage.

Finally, the role database table must be populated. Here, the `manager` role is added, as it grants access to the example application:

```
INSERT INTO user_roles (login, role) VALUES ('maharaja', 'manager');
```

Adding a Tomcat User to MySQL

Tomcat must be given a username and password in `$CATALINA_HOME/conf/server.xml` that can be used to connect to the database in a JDBC Realm. Therefore, this user must be in the `mysql.user` table. The best way to create a new user is to use the `GRANT` command.

The `GRANT` command creates a user in the `mysql.user` table. MySQL uses this table to determine access privileges to its databases. It is important to restrict access to this table, because unlimited access would allow anyone to change the access rights to every database on the server.

The passwords should not be stored in cleartext, so it is fortunate that MySQL encrypts them automatically with `GRANT`. The following example creates a user called `tomcat` who will be accessing the database from the local machine with the password `tomcat`:

```
mysql> GRANT SELECT ON authority.*
    -> TO 'tomcat'@'localhost' IDENTIFIED BY 'tomcat';
mysql> FLUSH PRIVILEGES;
```

Here, `tomcat` is given `SELECT` privileges on all tables in the `authority` database. This access is sufficient for authentication, but real-world applications may well need more access. The `IDENTIFIED BY` clause specifies the user's password. MySQL automatically obfuscates this value and inserts it into the `user` table, as shown here:

```
mysql> SELECT Host, User, Password FROM mysql.user;
+---------------+----------+-------------------------------------------+
| Host          | User     | Password                                  |
+---------------+----------+-------------------------------------------+
| localhost     | tomcat   | *BC76B32594D63CEE07D4144CBFD349B88E2FDBBB |
+---------------+----------+-------------------------------------------+
```

Note that there will be a few other records in addition to the one shown. They don't concern us here.

The `SET` command can be used to change the password of a user without having to create it afresh:

```
SET PASSWORD FOR 'tomcat'@'localhost' = PASSWORD('new_password');
```

To confirm that the `tomcat` user has indeed been given the appropriate privileges on `authority`, the following query can be used (note the `authority` database in the `Db` column):

```
mysql> SELECT Host, Db, User, Select_priv FROM mysql.db;
+---------------+-----------+--------+-------------+
| Host          | Db        | User   | Select_priv |
+---------------+-----------+---- ---+---- --------+
| localhost     | authority | tomcat | Y           |
+---------------+-----------+--------+-------------+
```

Here, the `Y` in the `Select_priv` column indicates that the user in the `User` column (`tomcat`) has `SELECT` privileges on the table in the `Db` column (`authority`).

A user's privileges can be cancelled with the `REVOKE` command, as shown here:

```
mysql> REVOKE SELECT ON authority.* FROM 'tomcat'@'localhost';
mysql> FLUSH PRIVILEGES;
```

Now that a user for Tomcat has been created, the appropriate Tomcat Realm can be configured.

Defining the MySQL-Based JDBC Realm

To define the JDBC Realm, the default `UserDatabase` Realm must be disabled. To do so, comment out the following lines in the `server.xml` file:

```
<!--
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
   resourceName="UserDatabase"/>
-->
```

Next, define a JDBC Realm right after the line you just commented, mapping the tables and columns from the `authority` database:

```
<Realm className="org.apache.catalina.realm.JDBCRealm"
       driverName="com.mysql.jdbc.Driver"
       connectionURL="jdbc:mysql://localhost/authority"
       connectionName="tomcat" connectionPassword="tomcat"
       userTable="users" userNameCol="login" userCredCol="password"
       userRoleTable="user_roles" roleNameCol="role"
       digest="md5"/>
```

The `connectionURL` points to the database that contains the authentication details, which is accessed using the credentials supplied in the `connectionName` and `connectionPassword` attributes. The lines beginning with `userTable` and `userRoleTable` specify which tables in the database you should be

using to look up the user and role for authentication purposes. The `digest` attribute is the algorithm that Tomcat uses to digest the password entered by the user (in this case, MD5). As mentioned earlier, this attribute can be one of the two digest algorithms supported by `java.security.MessageDigest` (SHA or MD5).

The MySQL JDBC driver, Connector/J, must be installed in Tomcat for this Realm to function. This process is detailed in Chapter 13.

Testing the JDBC Realm

To see the Realm in action, start Tomcat and connect to the example Web application via the following URL:

```
http://localhost:8080/secure/
```

You should be presented with the login screen, as shown in Figure 14-4. Enter **maharaja** in the Username field and **tomcat** in the Password field.

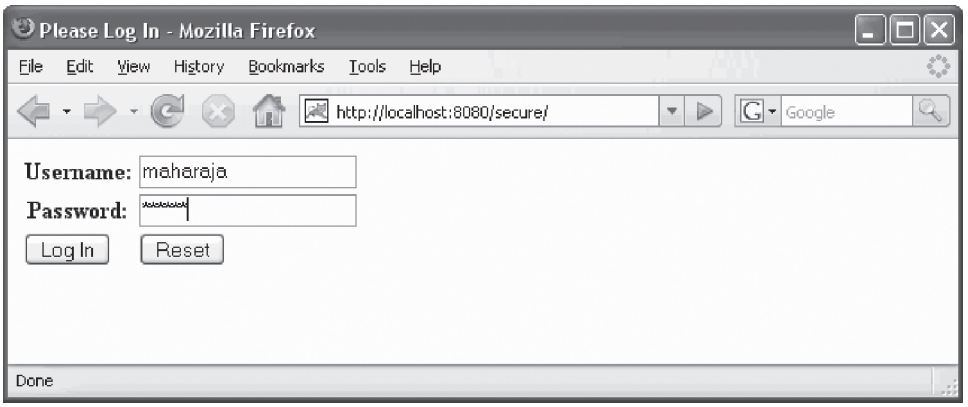

Figure 14-4: Login screen

Using the JDBC Realm, authentication is now performed against MySQL instead of against the `tomcat-users.xml` file (that is, the UserDatabase file-based Realm). By replacing the JDBC driver and changing the table/column mappings in the Realm configuration, other databases (for example, Oracle) with completely different schemata can be used. As long as the username, password, and role data are stored somewhere in the database, Tomcat can use this for authentication.

As demonstrated so far, changing the authentication method is easy and requires no code changes to the Web application. We also demonstrated custom login and authentication error forms and explained some methods for making the authentication process a lot more secure.

In some production scenarios, the user authentication and authorization information may not be available in a JDBC-accessible manner. The information may already be stored in directory services and/or external authentication and authorization systems. In some of these cases, configuring a JNDI Realm can enable Tomcat 5 to interoperate with the external systems. The next section describes JNDI Realms.

JNDI Realms

The Java Naming and Directory Interface (JNDI) is a standard Java API that provides applications with a unified interface to several different naming and directory services (such as SUN's NIS, Microsoft's ADS or NT Domains, and Novell's Netware Directory Service).

The JNDI architecture has two components: an API that is used by client-side applications to access the naming/directory services, and a Service Provider Interface (SPI), which allows vendors to develop custom Providers for their naming/directory servers. These Providers enable different directory servers to be "plugged in" in a manner transparent to the client application.

Lightweight Directory Access Protocol (LDAP) is one such directory protocol. OpenLDAP (`www.openldap.org`) and Netscape Directory Server (`http://enterprise.netscape.com/products/identsvcs/directory.html`) are two popular implementations of LDAP.

Further information on JNDI can be found at the following URL:

```
http://java.sun.com/products/jndi/docs.html
```

Similar to JDBC Realms, JNDI Realms enable you to use existing data in a directory service for a Realm. To use a JNDI Realm, you must be able to successfully map the various configuration attributes to an existing directory schema. This again is similar to the JDBC table and column name mapping. To better understand how this mapping works, the following table describes the configuration attributes that are available with a JNDI Realm.

Attribute	Description	Required?
`className`	Java programming class name of the JNDI Realm implementation. Must be set to `org.apache.catalina.realm.JNDIRealm`.	Yes
`connectionName`	The user name used to authenticate against the directory service via JNDI.	Yes
`connectionPassword`	The password used to authenticate against the directory service via JNDI.	Yes
`connectionURL`	The URL used to locate the directory service using JNDI.	Yes
`contextFactory`	Configures the Java programming language class used to create a context for the JNDI connection. The default LDAP-based factory is sufficient in all noncustom cases.	No
`Digest`	Specifies the digest algorithm used to store a password. By default, passwords are stored as plaintext.	No
`userPassword`	Maps the name of the directory attribute from the user element that contains the password information.	Yes
`userPattern`	Specifies an LDAP pattern for searching the directory for selecting user entry. Use the `{0}` as a placeholder for the distinguished name.	Yes
`roleName`	Maps the name of the directory attribute that contains the role name.	Yes

Table continued on following page

Attribute	Description	Required
`roleSearch`	Specifies an LDAP pattern for searching the directory for selecting roles entry. Use the `{0}` as a placeholder for the distinguished name, or `{1}` as a placeholder for the user name.	Yes
`roleBase`	Specifies the base element for role searches. The default is the top-level element.	No
`roleSubtree`	The default is `false`. If set to `true`, a subtree search will be conducted for the role.	No

The configurable attributes reveal that the username must map to individual elements at the top-level directory context. Each group of users assigned to the same role must also map to the individual element at the top-level directory context.

Configuring the JNDI Realm

A JNDI Realm stores data in an LDAP directory server (such as Netscape Directory Server, OpenLDAP, and so on) and accesses it using a JNDI Provider.

This configuration example uses OpenLDAP as the directory server. You can download OpenLDAP from `www.openldap.org/software/download/`. It is available in open source under the OpenLDAP Public License (`www.openldap.org/software/release/license.html`).

Coverage of OpenLDAP is beyond the scope of this chapter. On Linux distributions, you are likely to find a precompiled OpenLDAP module that you can install and get running.

For Windows, you may need to look around for OpenLDAP binaries and ports. One source for download of a working port is:

```
http://lucas.bergmans.us/hacks/openldap/
```

You can find information on LDAP at the following Web sites:

- **OpenLDAP, A Quick Start Guide:** `openldap.org/doc/admin/quickstart.html`
- **OpenLDAP 2.1, Administrator's Guide:** `openldap.org/doc/admin/`
- **Win32, Binaries Quick Start Guide:** `http://mguessan.free.fr/nt/openldap_en.html`

Configuring a JNDI Realm is more complex than configuring UserDatabase or JDBC Realms. The configuration involves a five-step process, as described in the following sections.

Installing the JNDI LDAP Driver

Place the JNDI LDAP driver JAR file in the `TOMCAT_INSTALLATION/lib` directory. The JNDI driver JAR file is typically named `ldap.jar`, is part of the LDAP provider download, and can be downloaded from `http://java.sun.com/products/jndi/`.

Creating the LDAP Schema

After installing the JNDI driver, you create the LDAP schema for storing the user and role data. This step is different for each directory server: Refer to your LDAP server documentation for further information.

Before creating the schema, there are some design issues to be considered. Connections to the directory server can be made either anonymously or by using the username and password specified in the Realm configuration by the `connectionName` and `connectionPassword` properties (see the section "Configuring the Realm," later in this discussion). An anonymous connection is sufficient in most cases.

Authentication of a user by a directory server can be done in two "modes": *bind mode* and *comparison mode:*

- **Bind mode:** In bind mode, user authentication is done by "binding" to the directory server using the *distinguished name (DN)* of the user and the password presented by the user. If the bind succeeds, the user is considered authenticated.

  Thus, in bind mode, the directory server does the actual authentication. The directory server saves a digested version of the user's password, and it converts the user's password to its digested version before comparing it. Therefore, the `digest` attribute in the Realm configuration in `server.xml` is ignored. However, this means that the password is transmitted as cleartext from Tomcat to the directory server. This is not the same as transmitting the password from the user's browser to the Tomcat end. Here, mechanisms such as HTTP DIGEST or even HTTPS may be used. Several LDAP servers support SSL connections, so this can be used to protect the transmission of passwords as cleartext.

- **Comparison mode:** In comparison mode, the Realm retrieves the password from the directory and does the comparison of the passwords itself. To enable comparison mode, you must specify the `userPassword` attribute of the Realm directive to the directory attribute that contains the user's password.

Bind mode is more secure because in comparison mode the configuration enables the Realm to read the user's password.

Another disadvantage of comparison mode is that the Realm implementation must handle password digests (in case the directory server stored the digested version of the password) and all the variations of the digest algorithms.

There are two approaches to storing roles in the JNDI directory:

- **Explicit directory entries:** Roles can be represented as explicit directory entries. In this case, the `roleBase`, `roleSubtree`, `roleSearch`, and `roleName` attributes in the Realm directive are used. These are discussed in more detail in the section "Configuring the Realm," later in this chapter.

- **Attributes of the user entry:** Alternatively, roles can also be represented as attributes in the user's LDAP directory entry. In this case, the `userRoleName` attribute (discussed later) in the Realm configuration should be set appropriately.

Populating the Directory

Now you are ready to populate the LDAP directory with the users for the `admin` and `manager` roles. This is required to use the `admin` and `manager` Web applications.

The following shows sample entries for the `admin` and `manager` roles, and two users (user1, and user2). user1 is listed in both roles:

```
# Define top-level entry
dn: dc=wrox,dc=com
objectClass: dcObject
objectClass: organization
dc: wrox
o: wroxpress
# Define an entry to contain people
# searches for users are based on this entry
dn: ou=people,dc=wrox,dc=com
objectClass: organizationalUnit
ou: people
# Define a user entry for user1
dn: uid=user1,ou=people,dc=wrox,dc=com
objectClass: inetOrgPerson
uid: user1
sn: user
cn: first user
mail: user1@wrox.com
userPassword: user1
# Define a user entry for user2
dn: uid=user2,ou=people,dc=wrox,dc=com
objectClass: inetOrgPerson
uid: user2
sn: user
cn: second user
mail: user2@wrox.com
userPassword: user2
# Define an entry to contain LDAP groups
# searches for roles are based on this entry
dn: ou=groups,dc=wrox,dc=com
objectClass: organizationalUnit
ou: groups
# Define an entry for the "admin" role
dn: cn=admin,ou=groups,dc=wrox,dc=com
objectClass: groupOfUniqueNames
cn: admin
uniqueMember: uid=user1,ou=people,dc=wrox,dc=com
uniqueMember: uid=user2,ou=people,dc=wrox,dc=com
# Define an entry for the "manager" role
dn: cn=manager,ou=groups,dc=wrox,dc=com
objectClass: groupOfUniqueNames
cn: manager
uniqueMember: uid=user1,ou=people,dc=wrox,dc=com
```

The previous data is in LDIF format, and it can be uploaded into OpenLDAP using the `ldapadd` tool:

```
$ ldapadd -f tomcat.ldif -x -D "cn=root,dc=wrox,dc=com" -w secret
```

In this example, `tomcat.ldif` is the file that contains the data about the roles and users in LDIF format. The value passed via the `-D` flag is that of the distinguished name used to bind to the directory server. This is the DN of a super-user who has the right to update the LDAP directory, and it authenticates itself

via a password (the `-w` option). This super-user was configured in the `rootdn` directive in `slapd.conf`. The following is a sample entry:

```
database ldbm
suffix dc="wrox",dc="com"
rootdn "cn=root,dc=wrox,dc=com"
rootpw secret
```

In order to successfully add the preceding LDIF file, you must ensure that the OpenLDAP server has successfully loaded the schema named `inetorgperson.schema`. *Check OpenLDAP documentation for more information on schema loading.*

Configuring the Realm

The Realm directive varies depending on how users bind to the LDAP directory. The following sample configuration is for an LDAP server running on the same machine (hence the `localhost` in the `connectionURL`), and has users logging in using a user ID (see the `userPattern` attribute specifying this):

```
<Realm   className="org.apache.catalina.realm.JNDIRealm"
        connectionURL="ldap://localhost:389"
        userPattern="uid={0},ou=people,dc=wrox,dc=com"
        roleBase="ou=groups,dc=wrox,dc=com"
        roleName="cn"
        roleSearch="(uniqueMember={0})"
/>
```

Finally, restart Tomcat 6 to make it reread the Realm configuration.

Adding Roles and Users

You can add a role or a user using the `ldapadd` command as discussed earlier in the section "Populating the Directory."

Other LDAP implementations (such as Netscape Directory Server) have GUI-based interfaces that make this simpler.

Removing a Role or a User

The `ldapremove` command is used to remove a user or role from the LDAP database:

```
$ ldapremove  "uid=user1,ou=people,dc=wrox,dc=com" -x -D
"cn=root,dc=wrox,dc=com" -w password
```

Here `"uid=user1,ou=people,dc=wrox,dc=com"` is the distinguished name of the user being deleted. The same command works for removing a role. You just specify the DN of the role to be deleted.

As before, the value passed via the `-D` flag is that of the distinguished name used to bind to the directory server. This is the DN of a super-user who has rights to update the LDAP directory, and it authenticates itself via a password (the `-w` option).

The next section shows how to work with a JAAS Realm.

JAAS Realm

The JAAS Realm uses the Java Authentication and Authorization Service (JAAS) to authenticate a user and provide access control.

JAAS enables the use of Pluggable Authentication Modules (PAM). With PAMs, the authentication technology is abstracted out, and thus the backend authentication technology can be rendered transparent to the application making the request.

Following is some of the basic terminology relevant to JAAS Realms. This is not a tutorial on JAAS; further information, including Javadocs and downloads, can be found at the following URL:

```
http://java.sun.com/products/jaas/
```

- **Subject:** The Subject (`javax.security.auth.Subject` class) is the identity that you wish to authenticate.
- **Principal:** The Principal (`java.security.Principal`) represents the interaction of a Subject with an authenticating authority.
- **LoginContext:** This is a Java class that acts as a session with the authentication Provider. It also loads the Provider class after reading its configuration file.
- **Provider:** This is a class that implements the `javax.security.auth.spi.LoginModule` interface, and contains the code for the actual authentication strategy.

JAAS is bundled-in with Java SE 6.

Configuration of a JAAS Realm

Configuring JAAS Realms is a five-step process:

1. Perform the setup required for the actual authentication technology.
2. Write or obtain a Provider for the authentication technology.
3. Configure the Provider.
4. Make changes to the Java security policy (if required).
5. Configure the Realm directive.

Performing the Setup Required for the Actual Authentication Technology

JAAS provides an API interface to the authentication technology. You first need to perform setup steps, if required, for this. For example, if you were using JNDI at the backend, you would need to install and configure a JNDI directory server.

Writing or Obtaining a Provider for the Authentication Technology

The Provider (discussed earlier) is a Java class that implements the `javax.security.auth.spi.LoginModule` interface. A Provider must implement the methods of this interface — namely, those shown in the following table.

Method	Description
`initialize`	Initializes the LoginModule
`abort`	Aborts the authentication process
`commit`	Commits the authentication process
`login`	Authenticates a Subject
`logout`	Logs out a Subject

The Provider would also make use of a `Principal` class (an implementation of the `java.security.Principal` interface) that represents users and roles in this particular implementation. For example, JAAS comes with implementations for Windows NT users and domains (`com.sun.security.auth.NTUserPrincipal` and `com.sun.security.auth.NTDomainPrincipal`).

JAAS also provides some Provider implementations as a part of the `jaasmod.jar` JAR file. These include a JNDI Provider (`com.sun.security.auth.module.JndiLoginModule`), an NT Login Provider (`com.sun.security.auth.module.NTLoginModule`), and a Solaris Login Provider (`com.sun.security.auth.module.SolarisLoginModule`).

In some cases, third-party vendors also provide Providers for their products.

Configuring the Provider

You must add configuration statements for the Provider in a configuration file. For some Providers (such as the Solaris and NT Login Providers), this is a very simple setup. The following code is a sample of the JAAS Provider configuration for Solaris's Login Provider:

```
SolarisLogin {
  com.sun.security.auth.module.SolarisLoginModule required;
};
```

Other Providers (such as the JNDI Provider) have a more complex setup (see the following sample). In general, the configuration attributes are Provider-specific:

```
JNDILogin {
  com.sun.security.auth.module.JndiLoginModule required
  user.provider.url="ldap://localhost:389/ou=People,dc=companyname,dc=com"
  group.provider.url="ldap://localhost:389/ou=Group,dc=companyname,dc=com";
};
```

The configuration for the Provider is passed to the JRE through the `java.security.auth.login.config` environment parameter.

Making Changes to the Java Security Policy (if Required)

The JAAS authentication Provider class is a trusted part of the system, and hence requires special access permissions. The following code is a sample Java policy file that shows the kind of permissions required:

```
//trust the Provider
grant codeBase "file:./provider/" {
  permission java.security.AllPermission;
};
```

(continued)

```
//trust JAAS
grant codeBase "file:/path/to/jaas.jar" {
  permission java.security.AllPermission;
};
//these permissions are needed by the client
grant codeBase "file:./client/" {
  permission javax.security.auth.AuthPermission "createLoginContext";
  permission javax.security.auth.AuthPermission "doAs";
  permission java.util.PropertyPermission "user.home", "read";
};
grant Principal com.sun.security.auth.NTUserPrincipal "user1" {
   permission java.util.PropertyPermission "user.home", "read";
};
```

This policy file is passed to the JRE through the `java.security.policy` environment parameter.

Configuring the Realm Directive

The following table describes the configuration attributes for the JAAS Realm element.

Attribute Name	Description	Required?
`className`	This is the class name of the java class that implements JAAS Realms. This must be `org.apache.Catalina.realm.JAASRealm`.	Mandatory
`debug`	The debug level. A missing or `0` (zero) valued debug level turns off debugging.The log file to which log messages are sent is specified in a Logger directive.	Optional
`appName`	The application name passed to the JAAS `LoginContext`, which uses it to select the set of relevant LoginModules. This name should match the name of the enclosing block in the JAAS Provider configuration.	Mandatory
`roleClassNames`	Comma-delimited list of `javax.security.Principal` classes that represent security roles.	Mandatory
`userClassNames`	Comma-delimited list of `javax.security.Principal` classes that represent individual users.	Mandatory

A sample configuration directive from `server.xml` is shown here:

```
<Realm className="org.apache.catalina.realm.JAASRealm"
       appName="Tomcat"
       roleClassNames="com.wrox.APrincipalImpl"
       userClassNames="com.wrox.AnotherPrincipalImpl"/>
```

Tomcat must be restarted in order for the Realm configuration changes to take effect.

Adding or Deleting Users and Roles

Adding or removing users and roles in a JAAS Realm is specific to the backend technology being used for authentication. For example, if NT Realms are used, adding a user would be equivalent to creating a new NT login account.

Single Sign-on

If two or more Web applications deployed in Tomcat are configured to use authentication, the user is prompted to authenticate, even if the same user uses the same credentials for both applications.

Fortunately, a special Tomcat mechanism enables users to log in only once in such scenarios: the Single Sign-on Valve. For detailed information on using this Valve, see Chapter 6.

In the next section, you examine SSL, which adds another level of security to the sample application in this chapter by preventing prying eyes from looking at data in transit.

Encryption with SSL

Secure Sockets Layer (SSL) is a protocol that enables secure communication between clients and servers in a network environment. Originally developed by Netscape, it has since been adopted as an Internet standard. SSL enables the encryption of traffic between the client and the server, and also provides an authentication mechanism. (This was briefly described earlier in this chapter in the discussion about the HTTP client certificate).

The security protocols on which SSL is based are *public key encryption* and *symmetric key encryption*. In public key encryption, a pair of encryption keys is used to encode a message: One is a publicly available key, and the other is a private key that is not disclosed to anyone else. Clients who want to send a message to an application that has a known public key need to encrypt it with that key. Only the corresponding private key can then decrypt the message, and thus the transmission is secure. Symmetric key encryption, conversely, uses the same (secret) key for both encryption and decryption. This algorithm, however, needs a reliable way to exchange the secret key between the two end points in the transmission.

When a client opens an SSL connection with a server, an *SSL handshake* is performed. The procedure for an SSL handshake is as follows:

1. The server sends its digital certificate to the client. This contains the public key of the server, information about the server, the authority that issued the certificate to the server, and the validity of the certificate.
2. The client then authenticates the server based on the validity of the certificate and the trustworthiness of the authority that issued the certificate. Certificates issued by well-known and trusted *Certificate Authorities (CAs)*, such as VeriSign, are recognized by most Web browsers. If the certificate cannot be validated, the user is warned and can choose to either accept the certificate or deny it.
3. A session key is then generated and exchanged over the connection. At this point, the connection is secured by the public key encryption mechanism, and so the exchange is secure. The session key is a symmetric key and is used for the duration of the session to encrypt all subsequent data transmissions.

The server configuration may also require the client to present its own authentication. Later in this chapter, you see how the `clientAuth` Tomcat attribute is used to enable this feature. In this situation, another step is introduced in the SSL handshake. Such a requirement is not common, and is used only in some business-to-business application environments.

The HTTPS (HTTP over SSL) protocol, as the name suggests, uses SSL as a layer on top of HTTP. Transport Layer Security (TLS) is the IETF (Internet Engineering Task Force) version of the SSL protocol. It is defined by RFC 2246 (`www.ietf.org/rfc/rfc2246.txt`), and is intended to eventually supersede SSL.

Adding support for SSL or TLS in Tomcat is a four-step process:

1. Download and install an SSL implementation.
2. Create a certificate *keystore*, to which a self-signed certificate is added.
3. Obtain a certificate from a third-party CA such as VeriSign (`www.verisign.com/`) or TrustCenter.de (`www.trustcenter.de/`). The self-signed certificate created previously is used to generate a certificate-signing request.

   If Tomcat is being used in a test/development environment, you can skip this step. In production environments, a CA-signed certificate may be desirable so that users will be willing to accept the certificate.
4. Configure Tomcat for SSL.

JSSE

Java Secure Socket Implementation (JSSE) is Sun's implementation of the SSL and TLS protocols.

JSSE is bundled with Java SE 6 and has been a standard library since JDK 1.4.*x*.

Preparing the Certificate Keystore

JSSE uses a keystore for the storage and retrieval of certificates. The keystore is simply a file. The commands for preparing a certificate keystore are as follows:

On Windows:

```
C:\> %JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA
```

On Linux:

```
$ $JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
```

The `-genkey` option specifies that a key pair (private key and public key) must be created. This key pair is enclosed in a self-signed certificate. The `-keyalg` option specifies the algorithm (which in this case is RSA) to be used for the key pair. All keystore entries are accessed via unique aliases using the `-alias` option. Here, the alias is specified as `tomcat`.

The `keytool` command asks for a password. The password can be set to the value Tomcat expects by default (`changeit`) or some other value. If the password is something other than the default, Tomcat's

`keystorePass` attribute will need to be changed, as shown later. The keytool then asks you for several other inputs (see Figure 14-5).

The default name for the keystore file is `.keystore` and it is stored in the user's home directory. This directory will vary depending on the operating system. On Linux, the `keystore` file would need to be in `/home/[username]`. On Windows 2000/XP, the `keystore` file would be in `C:\Documents and Settings\[username]`. An alternative keystore filename can be specified using the `-keystore` option. The password can also be specified on the command line with the `-keypass` option. Both of these methods are shown here:

```
C:\> %JAVA_HOME%/bin/keytool -genkey -alias tomcat -keyalg RSA -keypass somepass
-keystore /path/to/keystorefile
```

Figure 14-5 shows the `keytool` command being run.

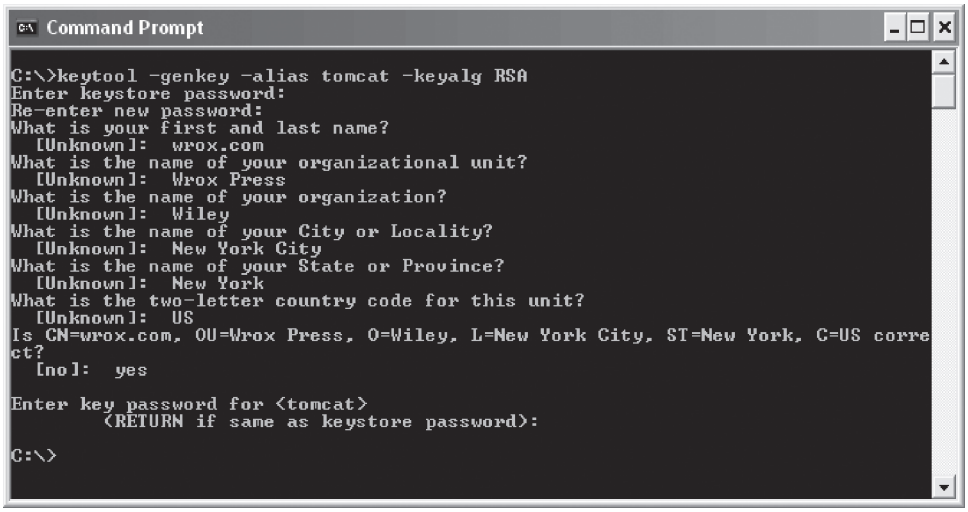
Command Prompt
C:\>keytool -genkey -alias tomcat -keyalg RSA
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: wrox.com
What is the name of your organizational unit?
[Unknown]: Wrox Press
What is the name of your organization?
[Unknown]: Wiley
What is the name of your City or Locality?
[Unknown]: New York City
What is the name of your State or Province?
[Unknown]: New York
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=wrox.com, OU=Wrox Press, O=Wiley, L=New York City, ST=New York, C=US correct?
[no]: yes
Enter key password for <tomcat>
(RETURN if same as keystore password):
C:\>

Figure 14-5: Creating the certificate keystore

Notice the Common Name (CN) field that has been entered as `wrox.com`. This must be of the format `www.domainname.com`, `hostname.domainname.com`, or just `domainname.com`. This name is embedded in the certificate. The CN should be the fully qualified host name for the machine on which Tomcat is deployed. If not, users will get a warning message in their Web browser when they try to access a secure page from Tomcat.

If this is a test/development environment, or a CA-issued certificate is not desired, the SSL setup is completed and now Tomcat-related setup changes must be performed.

The steps for obtaining a CA-signed certificate are covered in the next section.

Installing a Certificate from a Certificate Authority

To obtain a certificate from a CA, first a local certificate must be created using the `keytool` command:

```
$ keytool -genkey -alias tomcat -keyalg RSA
```

Next, this certificate is used to create a Certificate Signing Request (CSR):

```
$ keytool -certreq -keyalg RSA -alias tomcat -file certreq.csr
```

The `keytool` option (`-certreq`) creates a CSR file called `certreq.csr` that can be submitted to the CA to get a certificate. Figure 14-6 shows an example of this process.

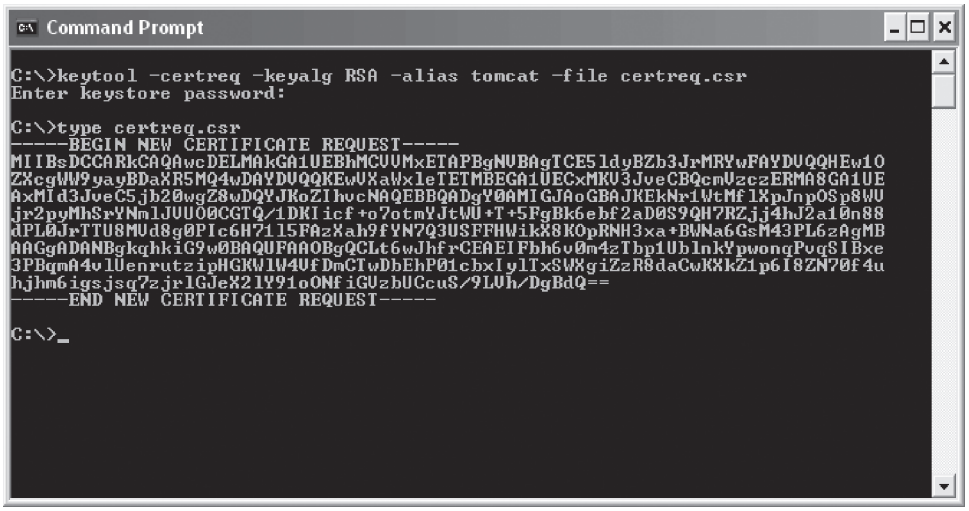

Figure 14-6: Generating the Certificate Signing Request

Obtaining a certificate requires payment to the CA for the authentication services. However, some CAs offer test certificates at no cost, although they are usually valid only for a limited time. To submit the CSR, visit VeriSign (`www.verisign.com`), Thawte (`www.thawte.com`), or TrustCenter.de (`www.trustcenter.de`).

After you have the certificate from the CA, you must get the Chain Certificate (also called the Root Certificate) from the CA. For VeriSign, this can be downloaded from the following site:

```
www.verisign.com/support/install/intermediate.html
```

The Chain Certificate is a self-signed certificate from the CA that contains its well-known public key. You can view the contents of a certificate using the `-printcert` option:

```
C:\> keytool -printcert -file /path/to/certificate
```

This is good practice before importing a third-party certificate into the keystore. You then import the Chain Certificate into the keystore:

```
C:\> keytool -import -alias root -trustcacerts -file
<filename_of_the_chain_certificate>
```

Here, the `<filename_of_the_chain_certificate>` contains the Chain Certificate that you got from the CA.

Finally, you import the new certificate:

```
C:\> keytool -import -alias tomcat -trustcacerts -file <your_certificate_filename>
```

In the next section, we examine Tomcat-related setup changes.

Protecting Resources with SSL

Resources can be protected with SSL just as they can be protected with authentication constraints. The `<user-data-constraint>` subelement of `<security-constraint>` in `web.xml` is used to specify the guaranteed integrity of the data flowing between the client and the server for this resource. There are three levels of integrity: `NONE`, `INTEGRAL`, and `CONFIDENTIAL`.

`NONE` means there is no guarantee that the data has not been intercepted and tampered with, while `INTEGRAL` guarantees the integrity of the data (meaning that the data has not been interfered with). The strongest guarantee is `CONFIDENTIAL`, which guarantees that a third party has not intercepted the data. If you specify `INTEGRAL` or `CONFIDENTIAL`, the server uses SSL for all requests to this resource by redirecting the client to the SSL port of the server. The redirection port is configured in the `redirectPort` attribute of the HTTP Connector.

For the `secure` application introduced earlier, the `CONFIDENTIAL` level is used. This is accomplished by adding the following element to the `<security-constraint>` in the example `web.xml` file:

```
<security-constraint>
  ...
  <user-data-constraint>
    <description>
     Constrain the user data transport for the whole application
    </description>
    <transport-guarantee>CONFIDENTIAL</transportÐguarantee>
  </user-data-constraint>
</security-constraint>
```

This forces all requests for the `secure` Web application to use HTTPS, even if the original came in over HTTP. This is the only setup required in `web.xml`. The next section considers changes to `server.xml`.

Tomcat Setup

The setup procedure for Tomcat 6 is straightforward. A handy HTTP Connector is already set up for SSL, and it needs only minor modification.

Locate the following `<Connector>` element in `server.xml`:

```
<!--
  <Connector
       port="8443"
       scheme="https" secure="true" SSLEnabled="true"
       keystoreFile="${user.home}/.keystore"
      clientAuth="false" sslProtocol="TLS"/>    -->
```

To use this Connector, the `<!--` and `-->` comment tags must be removed from around the `<Connector>` element. Next, if a nondefault password was used for the keystore (that is, any password but

"changeit"), the `keystorePass` attribute must be added to the `<Connector>` element containing the keystore password, as shown in bold in the following example:

```
<Connector
           port="8443"
           scheme="https" secure="true" SSLEnabled="true"
           keystoreFile="${user.home}/.keystore"
          sslProtocol="TLS"
           keystorePass="tomcat" />
```

Look at the top of the same `server.xml` file; you must make sure that APR optimization is not enabled. This can be done by setting `SSLEngine` to off in the `<Listener>` element.

```
  <Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="off" />
```

If you are using APR optimization and a native code SSL Engine (see Chapter 6 for more information on enabling native code SSL Engines), you need to change the `<Connector>` to:

```
<Connector protocol="org.apache.coyote.http11.Http11AprProtocol"
           port="8443"
           scheme="https" secure="true" SSLEnabled="true"
           SSLCertificateFile="/mycertdir/server.crt"
           SSLCertificateKeyFile="/mycertdir/ssl/server.pem"
           sslProtocol="TLS"/>
```

To test this feature, start Tomcat and request the following URL:

```
https://localhost/secure/
```

If a CA-signed certificate was not used, the browser displays a warning about the certificate (note that this warning may vary depending on your browser), as shown in Figure 14-7.

Website Certified by an Unknown Authority
Unable to verify the identity of wrox.com as a trusted site.
Possible reasons for this error:
- Your browser does not recognize the Certificate Authority that issued the site's certificate.
- The site's certificate is incomplete due to a server misconfiguration.
- You are connected to a site pretending to be wrox.com, possibly to obtain your confidential information.
Please notify the site's webmaster about this problem.
Before accepting this certificate, you should examine this site's certificate carefully. Are you willing to to accept this certificate for the purpose of identifying the Web site wrox.com?
Examine Certificate...
Accept this certificate permanently
Accept this certificate temporarily for this session
Do not accept this certificate and do not connect to this Web site
OK
Cancel

Figure 14-7: Warning about a certificate not signed by a Certificate Authority

You can click Examine Certificate to see the details of the certificate, or click OK and access the protected Web page via SSL.

Securing DefaultServlet

Tomcat uses a `DefaultServlet` to serve any static Web resources that do not map to a servlet (or JSP). This `DefaultServlet` is configured via the default `TOMCAT_INSTALLATION\conf\web.xml` deployment descriptor.

When a URL request is destined for a server-side directory, a welcome file (such as `index.xml`) is displayed. The exact welcome file displayed is controlled via the `<welcome-file-list>` element. The default list is as follows:

```
<welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

When one of the files in the preceding list cannot be found in the directory, the server can be configured to provide a directory listing by default.

The ability for users to see such directory listings can represent a potential security problem.

Disabling Directory Listing

By default, the `DefaultServlet` will not provide a directory listing if a designated welcome file cannot be found in a URL referenced directory. This behavior is configured through the initialization parameter to the servlet. You should double-check to make sure this listing is enabled. The highlighted `<init-param>` element in the following listing shows how to disable a directory listing:

```
    <servlet>
        <servlet-name>default</servlet-name>
        <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
        <init-param>
            <param-name>debug</param-name>
            <param-value>0</param-value>
        </init-param>
        <init-param>
            <param-name>listings</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
     </servlet>
```

Disabling an Invoker Servlet, SSI, and CGI Gateway

Tomcat 6 can support an Invoker servlet (to execute arbitrary non-preconfigured servlets for testing), Server Side Include (SSI), as well as CGI (ability to execute external logic coded in compatible languages). All of these features are controlled via the `web.xml` file and are disabled by default.

The invoker servlet should never be enabled in a production system. If the design or development team require the SSI or CGI functionality, make sure they are completely comfortable with the potential security risks associated with using such technology in production. Securing SSI- and CGI-based applications is a complex topic that is out of the scope of this book.

You should double check that these features are disabled in your production configuration. The following checks can be used:

- Examine the `$CATALINA_HOME/conf/server.xml` file and make sure that the CGI servlet, the SSI servlet, and the SSI filter are not enabled; their configurations should be commented out or removed altogether. See Chapter 5 for the location of these servlets and filters.
- Examine the `$CATALINA_HOME/conf/context.xml` and make sure that the `<Context>` element does not contain a `privileged='true'` attribute. This attribute must be removed or set to `false` for a secured set of applications.

Host Restriction

The last security mechanism covered in this chapter is perhaps one of the most effective and least complex: host restriction.

Rather than allow any user from any location in the entire Internet to use a Web application, system administrations may configure Tomcat to accept HTTP requests from either a specific IP address or a range of IP addresses. Requests from any other source will simply be ignored.

Tomcat restricts the hosts allowed to access an application through the use of the Request Filter Valve. For details, see Chapter 6.

Summary

This chapter has covered a broad range of techniques for securing Tomcat 6 itself and Tomcat-hosted applications. To conclude this chapter, let's review some of its key points:

- The MD5 hash and PGP signature of the downloaded Tomcat binary should be religiously checked to ensure their integrity.
- Unnecessary default Tomcat applications that may pose potential security risks should be disabled.
- The default Tomcat security-related settings should be changed as these could be used to attack the Web site.

- Tomcat should be run under a Tomcat-specific account with limited permissions.
- The Java Security Manager can be used to limit the operations that Web applications may perform.
- Web applications can be secured by using standard mechanisms for authentication and access control.
- The directory listing feature of DefaultServlet should be disabled.
- Invoker servlets should be disabled. SSI and CGI features should be disabled from the default Web descriptor if not absolutely required by the running applications.
- SSL can be used to secure important data communication between the Web server and the browser.
- Access to Web applications can be restricted to certain IP addresses or range of IP addresses.

The next chapter discusses shared hosting using Tomcat.

15

Shared Tomcat Hosting

Many small businesses do not need, or cannot afford, the cost of running a Web site hosted on a dedicated server or a cluster of servers, and hiring IT engineers to maintain them. A common solution for these businesses is to use a hosting service provided by an Internet Hosting Service Provider. Typically, these hosting services are shared hosting situations in which multiple Web sites can be running on a single computer. Running more than one Web site on one computer is called *virtual hosting*.

These hosting services allow for the sharing of resources such as the Web server, database server, mail server, firewall, and so on. Thus, all the services that are typically used in this scenario must have built-in support for shared hosting.

The following shared hosting topics are covered in this chapter:

- An introduction to virtual hosting terminology
- Virtual hosting using the Apache HTTP server
- Virtual hosting using Tomcat in both a standalone configuration, as well as with the Apache HTTP server
- Options for tuning Tomcat resource usage in a shared hosting situation

Apache 2.2, Tomcat 6, and `mod_jk` are used for all of the examples in this chapter.

Virtual Hosting Concepts

In this chapter, the term "Web site" refers to the contents of a distinct *Fully Qualified Domain Name (FQDN)*, which is served by a Web server. Strictly, a FQDN consists of two parts: a host name and a domain name. For example, the FQDN `www.wrox.com` consists of the host name `www` and the domain name `wrox.com`. The domain name `wrox.com` has other hosts such as `p2p` and

`newsletter`, whose FQDNs would be `p2p.wrox.com` and `newsletter.wrox.com`. However, because the distinction between an FQDN and a domain name is not relevant in this discussion, the terms are used interchangeably.

Typically there are two ways in which virtual hosting is implemented: *IP-based virtual hosting* and *name-based virtual hosting*.

- **IP-based virtual hosting:** In this mechanism, each Web site domain needs to have a different IP address. The Web server listens to each of these network interfaces, and serves resources from the relevant domain based on the IP address.
- **Name-based virtual hosting:** The Web site domains can share an IP address, and the Web server determines which domain the request is for based on the HTTP request headers.

Each of these mechanisms has advantages and disadvantages. IP-based virtual hosting requires either *multihoming hosts* — machines with multiple network interface cards (NICs) — or setting up virtual network interfaces. This often runs into limits, and besides IP addresses are a scare resource.

Name-based virtual hosting overcomes these issues, but has its own set of limitations. For instance, SSL requires unique IP addresses, so secure Web sites cannot use name-based virtual hosting. Also, some really, really old browsers don't support sending the extra HTTP request header information required to support name-based virtual hosting.

Virtual Hosting in Apache

This section covers how IP- and name-based virtual hosting is configured in the Apache HTTP server. A common configuration for Web deployments is to have Apache serve up requests for static resources (HTML pages, images, multimedia), and have Tomcat handle requests for JSPs and servlets. In such environments, it is important to understand how Apache is configured for virtual hosting.

It should be noted that the choice of IP or name-based virtual hosting is not an either-or choice: Apache is quite capable of supporting both of these in the same configuration.

Example Deployment Scenario

Before you continue configuring Apache for virtual hosting, look at the example deployment scenario that will be used in the rest of the chapter.

This section covers configuring Apache to serve two fictitious virtual hosts: `europa.dom` and `callisto.dom` on the same Apache instance. To ensure that these domains are really fictitious, we have even chosen a fictitious TLD (top level domain) name "dom." In this example, we add our own DNS entries for the virtual hosts, so using a fictitious TLD is not a problem. Naturally, these virtual hosts will be running on the same IP address for name-based virtual hosting, and on different IP addresses for IP-based virtual hosting.

Each of the domains would have its own document root in `/home/websites/<domain-name>/web`.

For the purposes of providing a simple test example, create a sample HTML page named `index.html` in the document root of each of these domains. For example, the file `/home/websites/europa.dom/web/index.html` should contain the following HTML:

```
<html>
  <head>
    <title>Europa.com's Apache website</title>
  </head>
  <body>
    Welcome to Europa.dom's Apache hosted website
  </body>
</html>
```

Create a similar file as `/home/websites/callisto.dom/web/index.html`, and change `europa.dom` to `callisto.dom`.

You also need to ensure that the domains `europa.dom` and `callisto.dom` can be resolved by the client and the server machines. This should be done by setting up your DNS server entries for these hosts. DNS configuration is not covered in this chapter, although for testing purposes, you can edit the hosts file on the server and client machine, as shown, to get the same effect:

For name-based virtual hosting, set `europa.dom` and `callisto.dom` to the same IP address (here 192.168.1.2), and add the following line to your hosts file:

```
192.168.1.2     callisto.dom europa.dom
```

For the IP-based virtual hosting example, change this so that the domains have different IP addresses (here 192.168.1.2 and 192.168.1.20):

```
192.168.1.2     callisto.dom
192.168.1.20    europa.dom
```

Naturally, these need to be valid IP addresses for the server machine. The next section explains how you can get another IP address when your server machine has just one network card.

IP-Based Virtual Hosting in Apache

In *IP-based virtual hosting*, a machine is configured to have a number of IP addresses equal to the number of hosts it will serve. Therefore, a machine that is to host ten Web sites would need ten IP addresses configured. These additional IP addresses may be configured either by adding physical network interfaces (NICs) — network cards — to the machine, or, as is more common, by adding aliased network interfaces to the computer.

Normally, when an NIC is added to a machine, it is configured with a single IP address, which is used by various services. However, it is possible to configure the same NIC with more than one IP address. Adding these additional IP addresses involves using operating system–specific commands for first creating a virtual interface and then configuring it with a virtual IP address. This process normally involves using a physical NIC and adding virtual interfaces on top of it, a process also commonly known as *aliasing*.

For example, on Linux, using the `ifconfig` command adds a virtual interface and configures the NIC with an IP address at the same time. If an Ethernet interface named `eth0` has already been configured, it is simple to add an aliased interface called `eth0:1` (in Linux, virtual Ethernet interfaces are named with the syntax `<physical-interface-name>:<virtual-interface-index>`), using the following command:

```
$ ifconfig eth0:1 <virtual-IP> netmask <virtual-IP-netmask>
```

Configuring IP-Based Virtual Hosting in Apache

Adding IP-based virtual hosts in Apache is trivial. Merely add a `<VirtualHost>` block to Apache's `httpd.conf` file for each corresponding Web site, and a few associated parameters. Let's look at a sample configuration:

```
Listen 80
...
<VirtualHost 192.168.1.2>
    ServerName europa.dom
    DocumentRoot /home/websites/europa.dom/web
    ServerAdmin support@europa.dom
    ErrorLog /home/websites/europa.dom/log/error
    TransferLog /home/websites/europa.dom/log/access
</VirtualHost>
<VirtualHost 192.168.1.20>
    ServerName callisto.dom
    DocumentRoot /home/websites/callisto.dom/web
    ServerAdmin support@callisto.dom
    ErrorLog /home/websites/callisto.dom/log/error
    TransferLog /home/websites/callisto.dom/log/access
</VirtualHost>
```

Here, two IP-based virtual hosts, `europa.dom` and `callisto.dom`, are configured to run on the IP addresses `192.168.1.2` and `192.168.1.20`, respectively. The `Listen` directive listed in the beginning causes Apache to listen on port 80 on all network interfaces — `192.168.1.2`, `192.168.1.20`, plus any others configured on this server machine. If you need to have it listen on specific network interface only, you can specify an IP address as a part of the directive as shown:

```
Listen 192.168.1.2:80
Listen 192.168.1.20:80
```

In this example, Apache is configured to listen on port 80; this means it needs to be started as root. To be able to run Apache as non-root, change the port number to 1024.

Each of the virtual hosts is defined in a `<VirtualHost>` section:

- The `ServerName` directive sets the domain name to be served by this virtual host.
- The `DocumentRoot` directive points to the base directory to be used for serving pages for this domain.
- The `ServerAdmin` directive lists the e-mail address of the Web server administration personnel.
- `TransferLog` and `ErrorLog` point to the log files to be used for Web site access and Web site error messages, respectively.

The two IP addresses used in the `<VirtualHost>` directives should belong to network interfaces for the machine on which Apache would be running. You may have noticed that each of the Web sites has its own document root and its own log files for access and error logging. Various other directives can be placed in these virtual host definitions to enable further customization. Omitting these other directives would cause the virtual host to inherit any values from the global settings in the configuration file.

Testing Your Configuration

Finally, the following summarizes the steps you need to take to configure IP-based virtual hosting on Apache:

1. Make DNS changes, or edit the host file for the hosts that you wish to run Apache on. For IP-based virtual hosting, each host must have a unique IP address. You can use the `ifconfig` command to add a virtual interface if your machine does not have multiple network cards.
2. Edit Apache's `httpd.conf` configuration file and add `Listen` directives for each of the virtual hosts, as shown in the previous section. Remove the existing `Listen` directive that listens on all network interfaces.
3. Edit the `httpd.conf` configuration file and add `VirtualHost` sections for each of the virtual hosts, as shown in the previous section.
4. Make sure that the `DocumentRoot` in each `VirtualHost` section points to the location of the Web site content for the virtual host.
5. Restart Apache using the `apachectl` command.

You should now be able to start your browser and view the respective Web sites. You can do this by either using the host names of the Web sites (`http://callisto.dom`, `http://europa.dom`) or even the IP addresses (`http://192.168.1.2`, `http://192.168.1.20`).

Avoiding Common Mistakes

Some common mistakes to avoid include the following:

- Apache, in its default configuration, starts up and listens on all the configured network interfaces on the machine. If, for some reason, Apache is configured to listen on only a restricted number of IP addresses on the machine (using the `Listen` directive), it is important to ensure that Apache is listening on all the IP addresses of the various IP-based virtual hosts in order for all of them to work.
- Using any random combination of IP addresses and Web host names will not always work as expected. This is commonly done by configuring the client machine to use a Domain Name Service (DNS) server. The Web client would query this DNS server for the IP address of the given host name, and then use the IP address returned by the DNS server to connect to the Web server. Similarly, the Web server would expect requests for the host name at the IP address specified in the corresponding `NameVirtualHost` directive.
- Needless to say, if the IP address given in the `NameVirtualHost` directive doesn't match the one returned by the DNS server for the host name, the Web client and the server won't be able to talk to each other.

- The FQDN of the Web site can be used in place of the IP address in the `<VirtualHost>` directive. In this case, there should not be any problems in the DNS resolution of the host names in the machine. This is because when Apache starts up, it resolves each of the FQDNs in its `<VirtualHost>` directives to their IP addresses before offering the Web service. Problems in resolving these addresses (for example, when a DNS server cannot be reached in time) during startup can cause Apache to abort prematurely.

Name-Based Virtual Hosting in Apache

While IP-based virtual hosts help maximize the use of resources, they are still not feasible in places where hundreds of domains must be hosted on the same machine. Obtaining one IP address for each host or configuring many network interfaces on the same machine becomes a logistical nightmare. In these cases, *name-based virtual hosting* can be used.

Name-based virtual hosting depends solely on an extension to the HTTP protocol. In an HTTP 1.0 protocol, a Web client or a browser merely had to make a TCP connection to port 80 of a Web server and request a document using a relative location identifier in order for the Web resource to be fetched. For example, to access the document `http://europa.dom/index.html`, the browser could look up the IP address of `europa.dom`, make a TCP connection to port 80 of the IP address, and get the complete resource just by using the HTTP `GET` command, as shown here:

```
$ telnet 192.168.1.2 80
Trying 192.168.1.2...
Connected to 192.168.1.2.
Escape character is '^]'.
GET /index.html HTTP/1.0
```

The response from the server would look as follows:

```
HTTP/1.1 200 OK
Date: Fri, 23 Feb 2007 07:38:44 GMT
Server: Apache/2.2.4 (Unix) mod_jk/1.2.20
Last-Modified: Fri, 23 Feb 2007 05:54:33 GMT
ETag: "10c07d-93-6d00d840"
Accept-Ranges: bytes
Content-Length: 147
Connection: close
Content-Type: text/html
[... Rest of the contents of index.html]
```

However, this enables only one Web site to be accessed per IP address; otherwise, it would be impossible to discover the host for which the request was intended.

To tackle this problem, the `Host:` header, as introduced in HTTP 1.1, is used to determine the Web site from which the resource is requested. This header has been implemented by many HTTP 1.0 clients, too. With this new header, the HTTP headers exchanged between an HTTP/1.1-compliant Web client and a server would look like the following from the client:

```
GET /index.html HTTP/1.0
Host: europa.dom
```

The additional `Host:` header in the client request helps the Web server distinguish between all the domains that share the same IP address.

Configuring Name-Based Virtual Hosting in Apache

Implementing name-based virtual hosting in Apache is not very different from implementing IP-based virtual hosting. It requires only the addition of the `NameVirtualHost` directive. This directive configures the IP address on which the Apache server will receive HTTP requests for the name-based virtual hosts. Documents should be subsequently fetched depending on the value of this parameter and the related virtual host definition specified later in the configuration.

A sample Apache name-based configuration for a Linux/Unix system would look like the following:

```
Listen 80
...
NameVirtualHost 192.168.1.2
<VirtualHost 192.168.1.2>
    ServerName europa.dom
    DocumentRoot /home/websites/europa.dom/web
    ServerAdmin support@europa.dom
    ErrorLog /home/websites/europa.dom/log/error
    TransferLog /home/websites/europa.dom/log/access
</VirtualHost>
<VirtualHost 192.168.1.2>
    ServerName callisto.dom
    DocumentRoot /home/websites/callisto.dom/web
    ServerAdmin support@callisto.dom
    ErrorLog /home/websites/callisto.dom/log/error
    TransferLog /home/websites/callisto.dom/log/access
</VirtualHost>
```

In this configuration, two Web sites, `europa.dom` and `callisto.dom`, are being hosted on the same IP address: `192.168.1.2`. After a request comes to the IP address, Apache uses the `Host:` parameter and the `ServerName` parameter of each of the virtual host definitions to determine the definition to which this request should be sent. The only configuration that must be specified in order to use name-based virtual hosting is to set up DNS settings for each of the FQDNs to be hosted so that the client can resolve the IP addresses correctly. Compare this to IP-based virtual hosting, whereby each of the IP addresses also had to be configured on the network interfaces of the machine.

You can even do name-based virtual hosts with the different sites running on different ports on the same IP address as shown:

```
Listen 80
Listen 8000
...
NameVirtualHost 192.168.1.2:80
NameVirtualHost 192.168.1.2:8000
<VirtualHost 192.168.1.2:80>
    ServerName europa.dom
    DocumentRoot /home/websites/europa.dom/web
    ServerAdmin support@europa.dom
    ErrorLog /home/websites/europa.dom/log/error
    TransferLog /home/websites/europa.dom/log/access
</VirtualHost>
```

(continued)

```
    <VirtualHost 192.168.1.2:8000>
        ServerName callisto.dom
        DocumentRoot /home/websites/callisto.dom/web
        ServerAdmin support@callisto.dom
        ErrorLog /home/websites/callisto.dom/log/error
        TransferLog /home/websites/callisto.dom/log/access
    </VirtualHost>
```

As mentioned earlier in the chapter, some old browsers do not send the `Host:` header required for name-based virtual hosting to work. There is a workaround in Apache for these browsers using the Apache `ServerPath` directive. To use this directive, add the following to the virtual host settings:

```
    NameVirtualHost 192.168.1.2
    <VirtualHost 192.168.1.2>
        ServerName europa.dom
        ServerPath /europa
        DocumentRoot /home/websites/europa.dom/web
        ServerAdmin support@europa.dom
        ErrorLog /home/websites/europa.dom/log/error
        TransferLog /home/websites/europa.dom/log/access
    </VirtualHost>
    ...
```

Once this is done, all requests for a Web page containing "`/europa`" can be accessed by older browsers (the ones that don't send the `Host:` parameter) at `http://europa.dom/europa`, and by the modern browsers at `http://europa.dom`. The Web developer would need to make sure that both locations work, for example by putting links between the two locations.

This is not a clean workaround; if an older browser doesn't send a `Host:` parameter and requests `http://callisto.dom/europa` for instance, Apache serves up content for the `europa.dom` Web site and not `callisto.dom`!

Testing Your Configuration

Finally, the following summarizes the steps you need to take to configure name-based virtual hosting on Apache:

1. Make DNS changes, or edit hosts file for the hosts that you wish to run Apache on. In name-based virtual hosting, different host names can share the same IP address.
2. Edit Apache's `httpd.conf` configuration file and add `NameVirtualHost` directives for each of the virtual hosts as shown in the previous section.
3. Edit the `httpd.conf` configuration file and add `VirtualHost` sections for each of the virtual hosts, as shown in the previous section.
4. Make sure that the `DocumentRoot` in each `VirtualHost` section points to the location of the Web site content for the virtual host.
5. Restart Apache using the `apachectl` command.

You should now be able to start your browser and view the respective Web sites. You can do this using only the host names of the Web sites (`http://callisto.dom`, `http://europa.dom`). If you use the IP

address (`http://192.168.1.2`), Apache is not able to resolve which Web site to serve and will serve you content from the default virtual host, which is the first one listed in the `httpd.conf` file — in this case, `europa.dom`.

Avoiding Common Issues

Some common issues to avoid in name-based virtual hosting include the following:

- If a Web request has been made to an IP address listed in the `NameVirtualHost` and the applicable virtual host could not be determined, Apache sends the request to the *first* virtual host block in the Apache configuration for that IP address. The request is *not* sent to the default document root of the whole server. Therefore, the first `<VirtualHost>` section for every `NameVirtualHost` IP address should be a domain where unresolved Web requests could also be handled.
- It is not possible to have SSL/HTTPS support using name-based virtual hosts because of the nature of the SSL protocol itself. For establishing the connection between the client and server, the SSL protocol parameters first need to be negotiated. For this to happen, the client has to know which server to connect to, which can't happen as the `Host:` header hasn't yet been exchanged to determine which virtual host to send the request to. This "chicken-and-egg" issue is the reason why each SSL-enabled Web site must be configured on a unique IP address. The only option here is to use IP-based virtual hosting.
- Older Web clients and many Web access software libraries still use the old HTTP 1.0 protocol. Because they don't send the `Host:` header to the Web server, name-based virtual hosting would not work properly with them. However, these incompatible clients are incredibly rare. It is unlikely that excluding them from a list of supported clients would cause a significant problem. All major browsers, such as Firefox, Netscape (version 2.0+), Internet Explorer (version 3.0+), and Lynx (1995+), support the `Host:` header.

Virtual Hosting in Tomcat

The preceding section explained how to configure Apache to support virtual hosts. This section contains the main focus of this chapter: virtual host support in Tomcat. Before reading further, it is important to consider what would be expected from Tomcat in a shared hosting environment.

> *Tomcat could work either in a standalone mode (in which it includes support for both an HTTP server and the JSP/Servlet container) or in a cooperative manner (with a Web server such as Apache). Chapter 11 provides details on various ways to set up Tomcat with the Apache Web server.*

Expecting Tomcat to provide virtual hosting support would mean the following: Given that two or more Web hosts are served from the same machine, when a request comes for a particular resource on one of these hosts, Tomcat should be able to successfully identify the host for which the request had been received, and fetch the required resource from the host document base.

For Tomcat working in a standalone mode, the request in question can target static pages, as well as JSP and servlets. When working along with another Web server such as Apache, the Web server itself handles the virtual hosts and the processing of subsequent static pages. Therefore, the only thing that

must be determined is whether Tomcat could handle the servlets and JSPs while distinguishing the various hosts involved.

Of course, the Apache Web server can be used to perform additional tasks such as load balancing and clustering. These configuration options are not generally considered for virtual hosting, and are discussed separately in Chapters 11 and 17.

The task of configuring virtual host support in Tomcat consists of two steps — adding a virtual host supporting Web application definition, which is sufficient if Tomcat is being run as a standalone server, and adding suitable directives in the Apache configuration file (`$APACHE_HOME/conf/httpd.conf`), if Tomcat is being run as an external Servlet engine. Let's look first at the scenario in which Tomcat is used as a standalone server, serving static pages as well as JSPs and servlets.

Example Deployment Scenario

Before you continue configuring Tomcat for virtual hosting, take a look at the example deployment scenario that will be used in the rest of the chapter.

This section covers configuring Tomcat to serve two fictitious virtual hosts: `europa.dom` and `callisto.dom` on the same Tomcat instance. To ensure that these domains are really fictitious, we have even chosen the fictitious TLD (top level domain) name "dom." In this example, we add our own DNS entries for the virtual hosts, so using a fictitious TLD is not a problem. Naturally, these virtual hosts will be running on the same IP address for name-based virtual hosting, and on different IP addresses for IP-based virtual hosting.

It is likely that in a production scenario, both these domains would be hosted on a directory outside the Tomcat base directory. The hosting scheme that is to be used is as follows.

Each of the domains would have its own document area in `/home/websites/<domain-name>`. Web applications or WAR files would be deployed in a subdirectory named `webapps`. Static HTML pages and scripts, if required, can be kept separate from the Web applications, and would be deployed in a subdirectory named `web`.

As an example for the domain `europa.dom`, a Web application called `shop` would be deployed under `/home/websites/europa.dom/webapps/shop`. Alternatively, the `shop.war` Web application archive could be deployed under `/home/websites/europa.dom/webapps/`. If the Web application needs to be available as the default context (i.e., `http://europa.com/` and not `http://europa.com/shop/`), then it needs to be deployed as `/home/websites/europa.dom/webapps/ROOT/`.

If Apache is required to serve static pages and scripts, they would be deployed to a separate location, which is used as the document root, such as `/home/websites/europa.dom/web/`. The static Web pages need not be served by Apache; they can very well be served by Tomcat itself. In that case, these are placed under `/home/websites/europa.dom/webapps/ROOT` (for the default context) or under `/home/websites/europa.dom/webapps/<context name>` (all other contexts).

However, in many shared hosting environments it makes sense to keep the Web applications separate from the static pages. This is because in a lot of shared hosting plans, Tomcat support is often an additional feature. For most of the clients' static site content, the separate Web directory would suffice, and Apache would handle them without any problems. For clients who want to add Java Web applications to their plan, it is a simple matter to deploy their WAR files in the `webapps/` directory without mixing them up with the static content.

Finally, keeping these two entities (static files and Web applications) separate aids in keeping the directory structure clean and more maintainable, as shown in Figure 15-1.

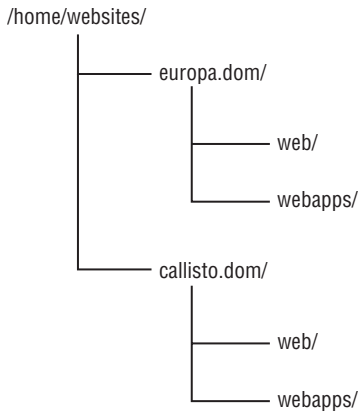

Figure 15-1: Sample directory structure for a virtual host environment

For the purposes of providing a simple test example, create a sample JSP file named `test.jsp`, in the document base of the default context Web application of each of these domains. For example, the file `/home/websites/europa.dom/webapps/ROOT/test.jsp` should contain the following simple code:

```
<html>
  <head>
    <title>Welcome to Europa!</title>
  </head>
  <body>
    <%
      out.println("Welcome to the Europa.dom web server.<br>");
      out.println ("Request sent to = "
                   + request.getServerName()
                   + ":" + request.getServerPort() + "<br>");
      out.println ("Request received by = "
                   + request.getLocalName()
                   + "[" + request.getLocalAddr() + "]:"
                   + request.getLocalPort() + "<br>");    %>
  </body>
</html>
```

Create a similar file as `/home/websites/callisto.dom/webapps/ROOT/test.jsp`. Remember to change the names for the `callisto.dom` domain.

Note the use of the `javax.servlet.ServletRequest` methods in the servlet, namely `request.getServerName()` and `request.getLocalName()`. These are for testing the script to ensure that the virtual host settings are correct. The `getServerName()` method returns the name of the server (or IP address) that the HTTP request was sent to. The `getLocalName()` method returns the name of the server that actually received the request. In this example, both are the same (`callisto.dom` and `europa.dom` respectively), but this need not always be the case. Also, as you will see in the section

"IP-Based Virtual Hosting in Tomcat," the `getLocalName()` method is used by Tomcat to map the request to a (virtual) host.

Feel free to change these setup details — for example, modifying the contents of the JSP or including additional Web application contexts to suit the server hosting policy required.

Tomcat as a Standalone Server

Figure 15-2 illustrates the relationship between the various components of Tomcat when Tomcat is being used as a standalone server.

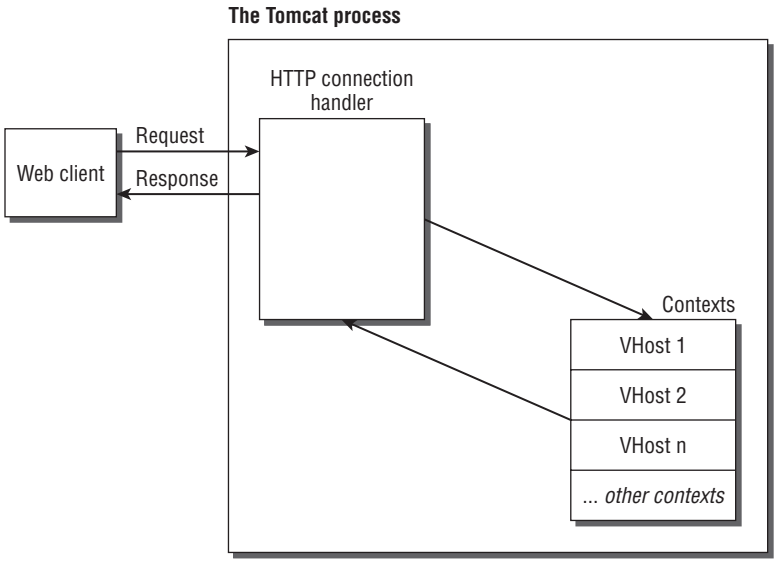

Figure 15-2: Tomcat handling a request for a virtual host

Name-Based Virtual Hosting in Tomcat

In Figure 15-2, the Web client directly sends the HTTP request to the Tomcat process listening at (default) port 8080. The HTTP Connector handles the Web client interaction. Tomcat then takes a look at the `Host:` header present in the HTTP request. If one is present, it tries to look up a virtual host with a name matching the one requested. If such a virtual host is found, the context parameter of the virtual host is taken and merged with the context parameters of the default configuration, and the file served accordingly. The resultant output is sent back to the Web client using the HTTP Connector again.

If no context with the given virtual host is found, Tomcat tries to match the context path to the contexts that do not belong to any virtual hosts. If one is found, that context is used to send back the results. If no such context is found, either the default (with the empty context path) context is used to send back the result (in which case, the context path is matched to a physical directory or file name) or an HTTP 404 error is generated and sent back to the client.

Adding a virtual host is as simple as adding an `<Engine>` entry, with `<Host>` entries for each additional virtual host in `server.xml`. Once you have made these additions, you simply restart Tomcat to use the virtual host definitions.

The default configuration of the top-level `<Service>` element in your `server.xml` file should look like the following:

```
<Server port='8005' shutdown='SHUTDOWN' debug='0'>
  ...
  <Service name='Catalina'>
  ...
  </Service>
</Server>
```

The rest of the configuration would be placed inside this `<Service>` container element. The next step is to add the Connectors to be used for this service. Because this is a standalone server, the only Connector required to be configured is the HTTP/1.1 Connector, to enable communication with the outside world. Check your Connector definition inside the `<Service>` element. An example Connection definition (this is the default definition) looks like the following one shown. This configures Tomcat to listen to port 8080 for incoming Web requests.

```
<Connector port="8080" protocol="HTTP/1.1"
           maxThreads="150" connectionTimeout="20000"
           redirectPort="8443" />
```

Now edit the `<Engine>` element enclosed within the `<Service>` element and set the default virtual host as shown:

```
<Engine name="Catalina"
        defaultHost="europa.dom"
        debug="0">
    ...
</Engine>
```

This specifies an Engine for the service that processes incoming requests from the Connectors. After any request is received by the Connector and passed on to the Engine, the Engine examines the HTTP headers (especially the `Host:` tag) to determine which of the virtual host definitions that it handles should receive the request. If none of the virtual hosts seems to match the request headers, the Engine passes on the request to a default host. The name of the default virtual host is specified in the attribute `defaultHost`. The value of this attribute must match a `<Host>` definition in the Engine.

In the previous configuration, the `defaultHost` property in the Engine element specifies that any Web requests that are not matched directly by the configured host elements should be served by the virtual host definition for `europa.dom`.

Adding the virtual host definition of the two hosts — `europa.dom` and `callisto.dom` — to the Engine is completed by including the following `Host` elements inside the `<Engine>` element:

```
<Engine name="Catalina" defaultHost="europa.dom" debug="0">
  ...
    <Host name="europa.dom" debug="0"
          appBase="/home/websites/europa.dom/webapps"
          autoDeploy="true"
          unpackWARs="true">
    </Host>
```

(continued)

```
        <Host name="callisto.dom" debug="0"
              appBase="/home/websites/callisto.dom/webapps"
              autoDeploy="true"
              unpackWARs="true">
        </Host>
      ...
      </Engine>
```

Logging functionality can be added to this virtual host by defining the `AccessLogValve` within the `<Host>` element:

```
        <Host name="europa.dom" debug="0"
            appBase="/home/websites/europa.dom/webapps"
            autoDeploy="true"
            unpackWARs="true">
            <Valve className="org.apache.catalina.valves.AccessLogValve"
                   directory="/home/websites/europa.dom/logs"
                   prefix="europa_access."
                   suffix=".log"
                   pattern="common"
                   resolveHosts="false"/>
        </Host>
```

`AccessLogValve` is covered in greater detail in Chapter 5. The directory specified for the log files in the Valve (i.e., `/home/websites/europa.dom/logs`) should be created if it does not exist.

Finally, the configuration is completed by (optionally) adding the contexts to serve for this virtual host, inside the `<Host>` element:

```
        <Host name="europa.dom" debug="0"
            appBase="/home/websites/europa.dom/webapps"
            autoDeploy="true"
            unpackWARs="true">
            ...
            <Context path="" docBase="ROOT" debug="0"/>
            <Context path="/shop" docBase="shop" debug="0"/>
            ...
        </Host>
```

This configuration has added two contexts here. The first one is the default context with an empty context path. This context has to be either defined explicitly or provided automatically by Tomcat (that is, without explicitly defining it in `server.xml`) if there is a Web application called `ROOT` in the `appBase` of the virtual host.

Adding contexts is optional: One convenience provided by Tomcat (version 4 and later) is that it creates automatic contexts if they exist in the `appBase`, even if you haven't defined them in the host definition. To provide this functionality, Tomcat looks at directories inside the `appBase` directory. If these directories follow the Web application structure (specifically, if they contain a `WEB-INF/web.xml` file in them), Tomcat automatically provides contexts with context paths equal to the name of the directory under `appBase`.

> *Remember that the default parameters for these contexts are picked up from* `$CATALINA_HOME/conf/web.xml`.

However, if you need to override some global parameters to these contexts, that configuration is required within the `<Context></Context>` elements. Examples include logging for this context in a separate file, context parameters, resource definitions, and so on.

The `server.xml` is not the only place where the context can be defined; you can also define it in a `context.xml` file, and place that file inside your Web applications `META-INF` directory. In this example, for instance, it would be at `/home/websites/europa.dom/webapps/ROOT/META-INF/context.xml` for the default (`ROOT`) context.

This completes the virtual host definition for `europa.dom`. For the virtual host `callisto.dom`, add another virtual host entry similar to that of `europa.dom`.

After completing these changes to `$CATALINA_HOME/conf/server.xml`, save the file and restart the Tomcat service.

Finally, set your DNS entries to point `europa.dom` and `callisto.dom` to the correct IP address of the server — the same in this case. For this test, instead of setting up an entry in a DNS server, you just add the following lines in your hosts file on the client and server machines:

```
192.168.1.2     callisto.dom europa.dom
```

Now, check the test JSP file in the `europa.dom` virtual host using the following URL, as shown in Figure 15-3.

```
http://europa.dom:8080/test.jsp
```

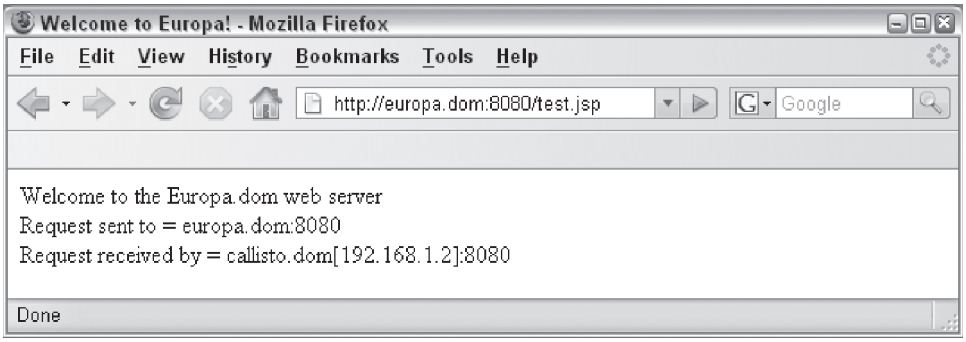
Welcome to Europa! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://europa.dom:8080/test.jsp
Google
Welcome to the Europa.dom web server
Request sent to = europa.dom:8080
Request received by = callisto.dom[192.168.1.2]:8080
Done

Figure 15-3: test.jsp executing in the Europa virtual host

As you can see from the "Welcome to the Europa.dom web server" text in Figure 15-3, the correct virtual host served up the JSP. The "Request sent to" text shows the result of the `request.getServerName()` method call, and that the request was indeed sent to `europa.dom`. Why then does the `request.getLocalName()` method call print out "Request received by = callisto.dom"? This is because the IP address maps to both `callisto.dom` and `europa.dom` in the hosts file (or DNS entry), and Tomcat resolves that to be one of them — in this case `callisto.dom`. This should be considered while developing a Web application to be deployed in shared hosting situations; the server name returned by the `javax.servlet.ServletRequest` methods may not always be what you expect.

Do the same for `callisto.dom` by using the following URL, as shown in Figure 15-4.

```
http://callisto.dom:8080/test.jsp
```

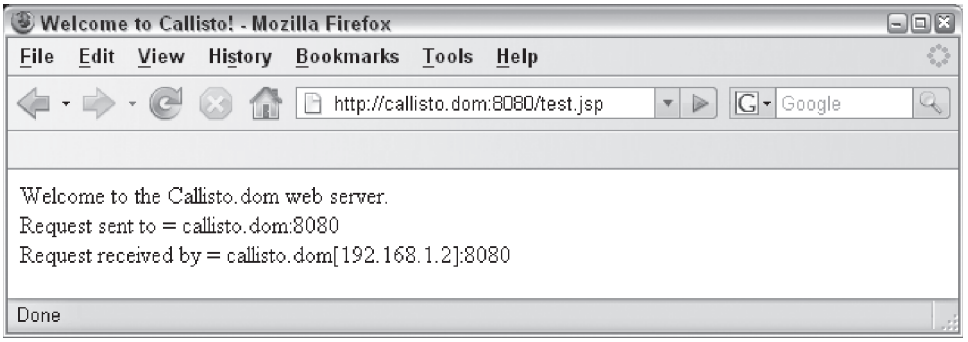
Welcome to Callisto! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://callisto.dom:8080/test.jsp
Google
Welcome to the Callisto.dom web server.
Request sent to = callisto.dom:8080
Request received by = callisto.dom[192.168.1.2]:8080
Done

Figure 15-4: test.jsp executing in the Callisto virtual host

Here, as you can see from the text "Welcome to the Callisto.dom web server" in Figure 15-4, the correct virtual host served up the JSP again. Note that the IP address (`192.168.1.2`) in Figures 15-3 and 15-4 are the same. In this case the "Request received by = callisto.dom" happens to print the correct host name, but that is just a coincidence.

Finally, perform a quick check to determine whether the default host setting of the `<Engine>` element is working properly. For this, use a host name other than the ones specified explicitly as `<Host>` definitions. The easiest way to do this is to try accessing the Tomcat server using the IP address `192.168.1.2` and view the results, as shown in Figure 15-5.

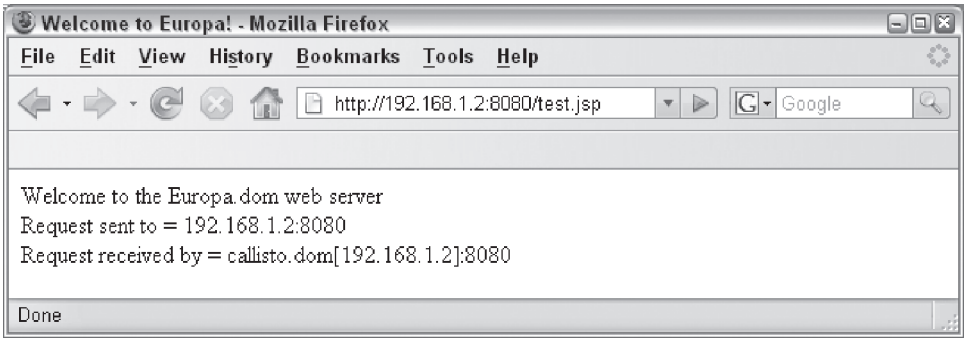
Welcome to Europa! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.2:8080/test.jsp
Google
Welcome to the Europa.dom web server
Request sent to = 192.168.1.2:8080
Request received by = callisto.dom[192.168.1.2]:8080
Done

Figure 15-5: test.jsp executing in the Europa virtual host, using the IP address

As shown by the "Welcome to the Europa.dom web server" text in Figure 15-5, Tomcat serves the contents of the `europa.dom` virtual host, as defined in the default virtual host entry of the Engine. Now that Tomcat is working as a standalone server for the virtual hosts, the next section provides instructions for creating the configuration to work with the Apache HTTPd.

The following list summarizes the steps you need to take to configure name-based virtual hosting in Tomcat (standalone):

1. Make DNS changes, or edit hosts file for the hosts that you wish to run Tomcat on. In name-based virtual hosting, different host names share the same IP address.
2. Create your deployment structure for the virtually hosted Web sites. This will be required when setting the `appBase` for the Web site's `<Host>` entries.
3. Edit Tomcat's `server.xml`, and modify the `<Engine>` directive, setting the default virtual host, as shown earlier.
4. Add `<Host>` entries inside the `<Engine>` directive for each virtual host.
5. Make sure that the `appBase` in each `<Host>` points to the location of the Web site content for that host.
6. Optionally, configure access log valves and contexts for each virtual host inside the `<Host>` entry.
7. Restart Tomcat. You should now be able to start your browser and view the respective Web sites. You can do this using only the host names of the Web sites (`http://callisto.dom:8080/test.jsp`, `http://europa.dom:8080/test.jsp`). If you use the IP address, Tomcat will serve you content from the default virtual host, as specified in your `<Engine>` directive.

IP-Based Virtual Hosting in Tomcat

Support for IP-based virtual hosting in a standalone Tomcat configuration is new; it was introduced in Tomcat version 5.5, and is not supported by versions earlier than that.

> *If you are testing name- and IP-based configurations with the same client machine, note that resolved IP addresses are often cached locally. Remember to clear the client cache before testing, or you might get spurious results. On Windows, for instance, the* `ipconfig /flushdns` *command clears out the local DNS resolver cache.*

The configuration required for using IP-based virtual hosting is similar to that of name-based, with one difference:

```
<Connector port="8080" protocol="HTTP/1.1"
           maxThreads="150" connectionTimeout="20000"
           redirectPort="8443"
           useIPVHosts="true"/>
...
<Engine name="Catalina" defaultHost="europa.dom" debug="0">
  ...

  <Host name="europa.dom" debug="0"
        appBase="/home/websites/europa.dom/webapps"
        autoDeploy="true"
        unpackWARs="true">
  </Host>
```

(continued)

```
        <Host name="callisto.dom" debug="0"
              appBase="/home/websites/callisto.dom/webapps"
              autoDeploy="true"
              unpackWARs="true">
        </Host>

        ...
      </Engine>
```

The Connector configuration needs the `useIPVHosts` setting to be set to `true` (the default is `false`). This causes Tomcat to use the server name to do the matching for the virtual host, not the passed `Host` attribute in the HTTP header.

To test this, configure your DNS entry for `callisto.dom` and `europa.dom` to different IP addresses. If your server machine has just one NIC card, you can add a new aliased interface, as shown in the following command for Linux:

```
$ ifconfig eth0:1 <virtual-IP> netmask <virtual-IP-netmask>
```

In the test setup, you just add entries in the `/etc/hosts` file for `callisto.dom` and `europa.dom`, setting them to two different IP addresses configured for the server machine:

```
192.168.1.2     callisto.dom
192.168.1.20    europa.dom
```

Now edit the Connector configuration in `server.xml`, set `useIPVHosts` to `true`, and restart Tomcat.

Then, as before, view the test JSP page for each of the domains. First view the test page for `callisto.dom` — browse to `http://callisto.dom:8080/test.jsp`, as shown in Figure 15-6.

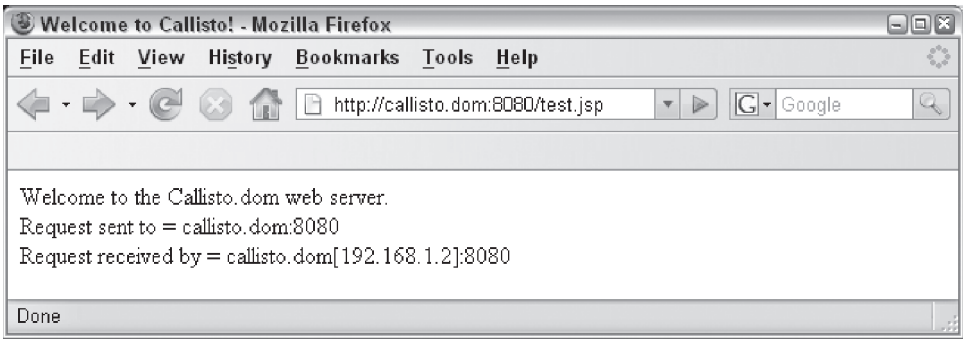
Welcome to Callisto! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://callisto.dom:8080/test.jsp
Google
Welcome to the Callisto.dom web server.
Request sent to = callisto.dom:8080
Request received by = callisto.dom[192.168.1.2]:8080
Done

Figure 15-6: test.jsp executing in the Callisto virtual host

Next, repeat this test for `europa.dom` by browsing to `http://europa.dom:8080/test.jsp`, as shown in Figure 15-7.

As you can see in Figures 15-6 and 15-7, the two virtual domains are at different IP addresses. This time around the correct host names are printed everywhere because there is just one virtual host per IP address.

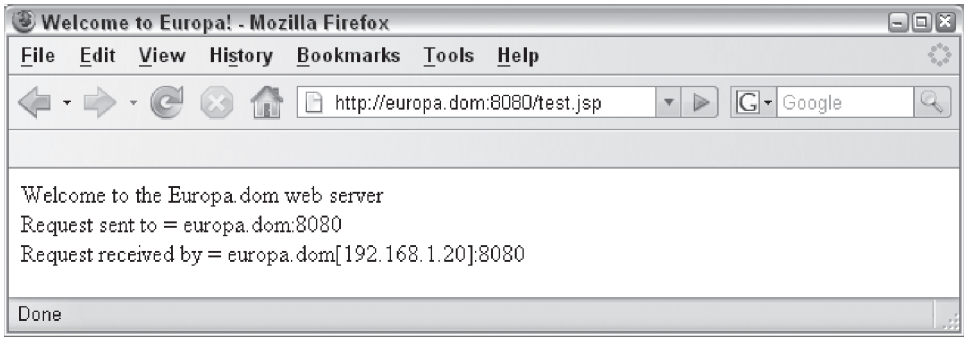
Welcome to Europa! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://europa.dom:8080/test.jsp
Google
Welcome to the Europa.dom web server
Request sent to = europa.dom:8080
Request received by = europa.dom[192.168.1.20]:8080
Done

Figure 15-7: test.jsp executing in the Europa virtual host

The same `useIPVHosts` parameter is used to support IP-based virtual hosting with an AJP Connector, too.

Finally, the following is a summary of the steps you need to take to configure IP-based virtual hosting in Tomcat (standalone):

1. Make DNS changes, or edit hosts file for the hosts that you wish to run Tomcat on. For IP-based virtual hosting, each host must have a unique IP address. You can use the `ifconfig` command to add a virtual interface if your machine does not have multiple network cards.
2. Create your deployment structure for the virtually hosted Web sites — this is required when setting the `appBase` for the Web site's `<Host>` entries.
3. Edit Tomcat's `server.xml`, and add the `useIPVHosts="true"`attribute to the Connector configuration.
4. Edit Tomcat's `server.xml`, and modify the `<Engine>` directive, setting the default virtual host, as shown earlier.
5. Add `<Host>` entries inside the `<Engine>` directive for each virtual host.
6. Make sure that the `appBase` in each `<Host>` points to the location of the Web site content for that host.
7. Optionally configure access log Valves and contexts for each host.
8. Restart Tomcat. You should now be able to start your browser and view the respective Web sites. You can do this using the host names of the Web sites as well as the IP addresses.

Tomcat with Apache

When Tomcat is used as an out-of-process servlet container along with Apache, two sets of configuration must be done: one in Tomcat and the other in Apache.

For Tomcat, the configuration shown in the preceding section remains more or less the same. The only difference is that you can disable the HTTP Connector in the `server.xml` configuration file because it is not being used.

To disable the HTTP Connector, either comment out or remove the default Connector setting (which follows) in `server.xml`:

```
<Connector port="8080" protocol="HTTP/1.1"
           maxThreads="150" connectionTimeout="20000"
           redirectPort="8443" />
```

The Connector that is being used is the `AJP/1.3` Connector that listens, by default, on port 8009:

```
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

At Apache's end, as shown in the previous chapters, a Connector for Tomcat, such as `mod_jk`, must be used.

The AJP protocol is covered in Chapter 4, and installing the `mod_jk` Apache module is covered in Chapter 11. The configuration you do is similar to that in Chapter 11, except for one difference: The `JKMount` directive is defined inside a `VirtualHost` directive in `httpd.conf`.

Figure 15-8 shows a diagrammatic representation of how the components are related. The differences between Figure 15-2 and Figure 15-8 reflect the different information pathways.

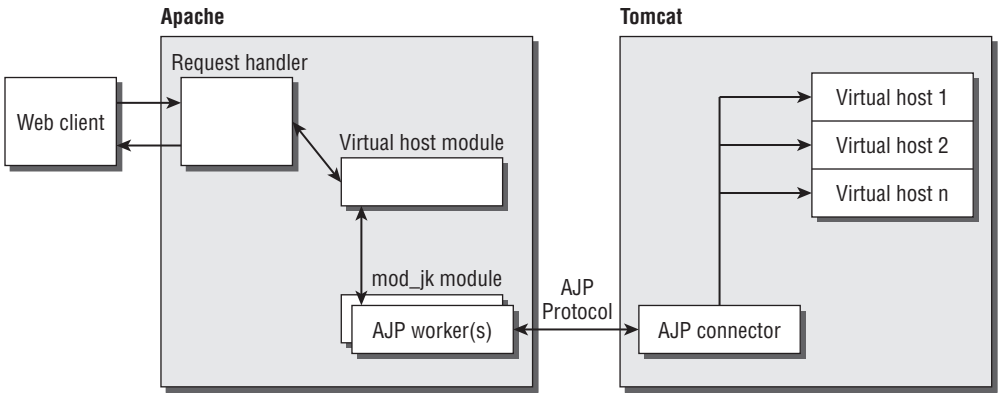

Figure 15-8: Apache and Tomcat serving HTTP requests

Here, Apache receives the HTTP request from the client. If name-based virtual hosting is used, it looks up the appropriate virtual host entry using the `Host:` parameter in the request. In the virtual host entry, `mod_jk` is configured to forward all appropriate servlet and JSP requests to the appropriate worker.

The worker would use the AJP protocol to communicate with a Tomcat process started within the Apache adapter. Tomcat then examines the request to determine whether any of its virtual host definitions match the request. This is similar to the matching process in the standalone Tomcat server. The servlet response is then sent back through the AJP Connector to the `mod_jk` module. This, in turn, instructs Apache to send the reply back to the Web client.

Configuring Apache

Assuming that `mod_jk` Apache module has been appropriately installed as explained in Chapter 11, you now take a look at adding virtual host support to this configuration.

As explained in the section "Name-Based Virtual Hosting with Apache," earlier in this chapter, for every virtual host definition you need to add a `<VirtualHost>` section in Apache (in fact, the Tomcat `<Host>` configuration definition is very similar to this concept). Now, along with the rest of the virtual host contents, you add some directives to connect certain resources to Tomcat.

Modify `httpd.conf`'s `<VirtualHost>` elements as follows:

```
Listen 80
...
# Load the mod_jk module
LoadModule    jk_module  modules/mod_jk.so
# Path to workers.properties file
JkWorkersFile /etc/httpd/conf/workers.properties
# Path to jk shared memory
JkShmFile     /var/log/httpd/mod_jk.shm
# Path to jk log file
JkLogFile     /var/log/httpd/mod_jk.log
# Set the jk log level [debug/error/info]
JkLogLevel    info
# Select the timestamp log format
JkLogStampFormat "[%a %b %d %H:%M:%S %Y] "
...
NameVirtualHost 192.168.1.2
<VirtualHost 192.168.1.2>
    ServerName callisto.dom
    DocumentRoot /home/websites/callisto.dom/web
    ServerAdmin support@callisto.dom
    ErrorLog /home/websites/callisto.dom/log/error
    TransferLog /home/websites/callisto.dom/log/access
    JkMount /*.jsp callisto-worker
</VirtualHost>
<VirtualHost 192.168.1.2>
    ServerName europa.dom
    DocumentRoot /home/websites/europa.dom/web
    ServerAdmin support@europa.dom
    ErrorLog /home/websites/europa.dom/log/error
    TransferLog /home/websites/europa.dom/log/access
    JkMount /*.jsp europa-worker
</VirtualHost>
```

The highlighted portions in the `httpd.conf` file show the common `mod_jk` settings (path to log file, shared memory file, `workers.properties` file, and so on) as well as the settings specific to each virtual host. These virtual host–specific settings include the `JkMount` directive that specifies which "worker" handles JSP pages for each virtual host.

The `workers.properties` file consists of the following:

```
# Define all the workers
worker.list=europa-worker, callisto-worker
# Set properties for callisto-worker
worker.callisto-worker.type=ajp13
worker.callisto-worker.host=callisto.dom
worker.callisto-worker.port=8009
```

(continued)

```
# Set properties for europa-worker
worker.callisto-worker.type=ajp13
worker.callisto-worker.host=europa.dom
worker.callisto-worker.port=8009
```

Restart the Apache HTTPd server and Tomcat and access the following previously used test URL:

```
http://europa.dom/test.jsp
```

However, this time, instead of sending the request to port 8080 of the Tomcat Web server, use the standard HTTP port 80 on which Apache should be listening. Notice how the page is rendered correctly. Of course, you should also be able to access the `http://europa.dom/index.html` page.

Repeat the test for the `callisto.dom` domain. In case of any error, look at the `mod_jk` log file defined in the `JkLogFile` directive in `http.conf` for error messages.

The previous settings were for name-based virtual hosting. For IP-based, things are very similar. The only changes are as follows:

- Allocate different IP addresses for the domains. Here, 192.168.1.2 and 192.168.1.20 are reserved for the two domains. This also involves doing the necessary DNS changes (or to a host file, as before).
- Comment or remove the `NameVirtualHost` directive in `httpd.conf`.

```
#NameVirtualHost 192.168.1.2
<VirtualHost 192.168.1.2>
    ServerName callisto.dom
    DocumentRoot /home/websites/callisto.dom/web
    ServerAdmin support@callisto.dom
    ErrorLog /home/websites/callisto.dom/log/error
    TransferLog /home/websites/callisto.dom/log/access
    JkMount /*.jsp callisto-worker
</VirtualHost>
<VirtualHost 192.168.1.20>
    ServerName europa.dom
    DocumentRoot /home/websites/europa.dom/web
    ServerAdmin support@europa.dom
    ErrorLog /home/websites/europa.dom/log/error
    TransferLog /home/websites/europa.dom/log/access
    JkMount /*.jsp europa-worker
</VirtualHost>
```

- Add the `useIPVHosts` attribute to the AJP Connector, and set it to `true`.

```
    <Connector port="8009" protocol="AJP/1.3" redirectPort="8443"
               useIPVHosts="true"/>
```

That's it. Now restart Apache and Tomcat, and test the URLs `http://europa.dom/index.html` and `http://europa.dom/test.jsp`.

For more details about connecting Apache and Tomcat, refer to Chapter 11.

The Tomcat Host-Manager Application

Tomcat 6 ships with a new — and not yet completely functional — management application called the `host-manager`. As the name suggests, this management application allows you to manage virtual hosts — create a new virtual host, delete an existing one, as well as list, start, and stop virtual hosts — without editing the `CATALINA_HOME/conf/server.xml` file.

This application is disabled by default. To enable it, you need to create a user with the `admin` role. The following shows the `CATALINA_HOME/conf/tomcat-users.xml` file with a user named tomcat, password tomcat, and with the roles of `manager` (required for the `manager` application) as well as `admin` (required for the `host-manager` application).

```
<tomcat-users>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="secret" roles="admin,manager"/>
</tomcat-users>
```

Once these changes are done and Tomcat has been restarted, you can access the host-manager application at the default URL `http://localhost:8080/host-manager/html/`.

The security considerations for using the host-manager are the same for the `manager` application as discussed in Chapter 8:

- Use a more rigorous mechanism of authentication for the `host-manager` application than `BASIC`.
- Use JDBC- or JNDI-based Realm implementations to store the manager username/password. These are more secure than Memory/UserDatabase Realms because they don't save the password in a text file on the file system.
- If you do use the Memory/UserDatabase, configure the Realm implementation to use encrypted passwords.
- Disable the `host-manager` application for production servers.
- If you do have to enable the `host-manager` application for production server for any reason, use a `RemoteAddrValve` or `RemoteHostValve` Valve to restrict the machines from which the `host-manager` application can be accessed.

Virtual Hosting Issues: Stability, Security, and Performance

After discussing the benefits of virtual hosting and how to set it up, it is only fair that you be aware of some of its drawbacks.

If the Tomcat instance is shared between virtual hosts, a badly written Web application can bring down the entire server, and thus bring down every Web site running on it.

Second, there is a lack of effective ways to control resource usage — memory, CPU, and disk space — by Web applications. As a system administrator, you can have each virtual host run in a different JVM, and

then put limits on the amount of memory used by the JVM, and that helps to some extent. This next section describes in greater detail how to do this.

Finally, if your Web site is hosted using name-based virtual hosting, you end up sharing your IP address with a lot of other Web sites that you have no control over. If one of those Web sites sends a lot of spam e-mail, your Web site's IP address could very well end up on a RBL (Real-time Blackhole List) — i.e., a blacklist for spammers. This would mean that you will have a lot of problems getting your e-mails out to their intended recipients. So select your Internet hosting service with care, and ask questions about the other Web sites they host.

Tuning Virtual Hosting Settings in Tomcat

The previous section listed some of the challenges posed by virtual hosting that relate to stability, security, and performance. Some of these issues can be addressed by the following configuration enhancements:

- ❑ Creating separate JVMs for each virtual host
- ❑ Setting memory resource limits for each Tomcat JVM
- ❑ Using the Java Security Manager to restrict what a Web application can do

Other than these Tomcat-specific configuration changes, system administrators for the hosting service providers should also ensure that file system permissions are set appropriately to prevent customers from being able to view or modify other Web sites' resources. This is especially important when the hosting service provides an `sftp` (secure ftp) account, a shell (`ssh`) account to customers to enable them to update their Web applications.

These mechanisms help provide a more stable and secure hosting environment in shared hosting situations.

Creating Separate JVMs for Each Virtual Host

The chapter has focused on how multiple hosts could be served from the same Tomcat process. While this would suffice for many providers, others would rightly raise the issue of security between the virtual hosts.

Because all the virtual hosts lie in the same request-processing Engine, trusted contexts in these virtual hosts (which can access Tomcat internal objects and load/unload other `webapps`, such as the default `manager` Web application) would have access to the common Tomcat internal classes and can hence encroach on one another's territory. This could be a logistical nightmare.

One possible solution would be to set up one `<Engine>` per virtual host in the same `server.xml` file. Because each `<Service>` container element in the file could have only one child `<Engine>` element, this would mean adding one service per virtual host, with the accompanying Engine. However, because every Service has its own set of Connectors, this would also mean setting up different Connectors listening on different ports for each Engine. Therefore, each virtual host in the Apache configuration would have to forward to a different worker.

While this removes the problem of sharing information between the virtual hosts, it still causes a bit of discomfort when you realize that a relaxed security policy in Tomcat can give one domain enough privileges to bring down the whole Tomcat process.

The more secure (albeit more resource-intensive) solution to such possible security problems is to have one Tomcat process per virtual host. Luckily, Tomcat has support for running multiple Tomcat processes using the same Tomcat binary installation.

Tomcat depends on two environment variables to find its internal classes. These variables are used to find the configuration-specific files, and are named `CATALINA_HOME` and `CATALINA_BASE`:

- `CATALINA_HOME` is needed for any Tomcat build to function properly. Tomcat uses this variable to find the location of internal classes and libraries.
- `CATALINA_BASE` is used by Tomcat to find the location of the configuration-specific files and directories, such as configuration files, the scratch directory in which JSPs are compiled, log files, and the various Web applications. In case `CATALINA_BASE` is not set, it defaults to the value of `CATALINA_HOME`.

Therefore, to have separate Tomcat processes, all that is required is to set the value of `CATALINA_BASE` to a different area of the disk for each virtual host, with its own `server.xml` file. This `server.xml` file would have only one virtual host definition, different Connector port numbers, and different directories for logs, scratch areas, and so on.

This setup requires the duplication of the directory trees that are specific to each implementation. While disk space is relatively cheap, this does introduce additional system administration overhead. Later in this chapter, we offer some simple scripts to illustrate how Tomcat server instances can be more easily managed.

For example, for the two virtual domains that you would be serving, you could store their respective configurations in two different directories under `/home/websites/<domain-name>/catalina`. For `europa.dom`, `CATALINA_BASE` could be `/home/websites/europa.dom/catalina`, and similarly, for `callisto.dom`, `CATALINA_BASE` could be set to `/home/websites/callisto.dom/catalina`. You would also need to create directories named `temp` and `log` under `/home/websites/<domain-name>/catalina` because Tomcat looks for these directories under `CATALINA_BASE`. `CATALINA_HOME` would still be common for both, say at `/usr/local/tomcat`.

The `server.xml` file of `europa.dom` located at `/home/websites/callisto.dom/catalina/conf/server.xml` should then be modified as follows:

- Change the `attribute` port of the `<Server>` root element that is used to shut down the Tomcat process so that it is unique and unused. It is set to `8105` in this example.
- In case of name-based virtual hosting — when the IP addresses are shared — change the AJP Connector port to a unique, unused port number, too — `8109` in this example. In case of IP-based virtual hosting, you can use the same port number as long you use the `address` attribute for the Connector, and set it to the IP address for `callisto.dom`. Also, for IP-based virtual hosting, remember to set the Connector's `useIPVHosts` attribute to `true`.
- The AJP Connector (shown previously) is used only if you have a combined Apache-Tomcat configuration. For a standalone Tomcat server, modify the non-SSL HTTP/1.1 Connector port to an unused unique port (here `8180`) for name-based virtual hosting. For IP-based virtual hosting, as before, you can use the same port number, so long you use the `address` attribute for the Connector, and set it to the IP address for `callisto.dom`. Again, set the `useIPVHosts` attribute to `true`.
- Ensure that only the virtual host definition of `europa.dom` is present, and that the default host of the `<Engine>` is set to this domain.

The relevant sections of the `europa.dom` `server.xml` file should end up looking like the following:

```
<Server port='8105' shutdown='SHUTDOWN' debug='0'>
  ...
  <Service name='Catalina'>
    <!-- A non-SSL HTTP/1.1, for a standalone Tomcat configuration -->
    <Connector port="8180" protocol="HTTP/1.1"
               maxThreads="150" connectionTimeout="20000"
               redirectPort="8443" />

    <!-- AJP/1.3 Connector for a configuration with Apache HTTP server -->
    <Connector port='8109'
               enableLookups='false' redirectPort='8443' debug='0'
               protocol='AJP/1.3' />
    <Engine name='Catalina' defaultHost='europa.dom' debug='0'>
      <Host name='europa.dom' debug='0'
          appBase='/home/websites/europa.dom/webapps'
          autoDeploy='true'
          unpackWARs='true'>

          <Valve className='org.apache.catalina.valves.AccessLogValve'
              directory='/home/websites/europa.dom/logs'
              prefix='europa_access.'
              suffix='.log'
              pattern='common' />
          <Logger className='org.apache.catalina.logger.FileLogger'
              directory='/home/websites/europa.dom/logs'
              prefix='europa_catalina.'
              suffix='.log'
              timestamp='true'/>

          <Context path='' docBase='ROOT' debug='0' />
      </Host>
    </Engine>
  </Service>
</Server>
```

For the `server.xml` file of `callisto.dom` located at `/home/websites/callisto.dom/catalina/conf/server.xml`, similar changes should be made:

- Change the `<Server>` port to a unique, unused port number — `8205` in this example.
- Again, in case of name-based virtual hosting — when the IP addresses are shared — change the AJP Connector port to an unused port number, too, such as `8209` in this example. For IP-based virtual hosting, you can use the same port number used for other virtual hosts, as long you use the `address` attribute for the Connector, and set it to the IP address for `callisto.dom`. Also, for IP-based virtual hosting, remember to set the Connector's `useIPVHosts` attribute to `true`.
- The AJP Connector (shown previously) is used only if you have a combined Apache-Tomcat configuration. For a standalone Tomcat server, modify the default non–SSL HTTP/1.1 Connector port to an unused unique port (here `8280`) for name-based virtual hosting. For IP-based virtual hosting, as before, you can use the same port number, as long as you use the `address` attribute for the Connector, and set it to the IP address for `callisto.dom`. Again, set the `useIPVHosts` attribute to `true`.

- Ensure that only the virtual host definition of `callisto.dom` is present, and that the default host of the `<Engine>` element is set to this domain.

The relevant sections of the `callisto.dom` `server.xml` file should end up looking like the following:

```
<Server port='8205' shutdown='SHUTDOWN' debug='0'>
   ...
  <Service name='Catalina'>
    <!-- A non-SSL HTTP/1.1, for a standalone Tomcat configuration -->
    <Connector port="8280" protocol="HTTP/1.1"
               maxThreads="150" connectionTimeout="20000"
               redirectPort="8443" />

    <!-- AJP/1.3 Connector for a configuration with Apache HTTP server -->
    <Connector port='8209'
               enableLookups='false' redirectPort='8443' debug='0'
               protocol='AJP/1.3' />
    <Engine name='Catalina' defaultHost='callisto.dom' debug='0'>
      <Host name='callisto.dom' debug='0'
          appBase='/home/websites/callisto.dom/webapps'
          autoDeploy='true'
          unpackWARs='true'>

          <Valve className='org.apache.catalina.valves.AccessLogValve'
              directory='/home/websites/callisto.dom/logs'
              prefix='callisto_access.'
              suffix='.log'
              pattern='common' />

          <Logger className='org.apache.catalina.logger.FileLogger'
              directory='/home/websites/callisto.dom/logs'
              prefix='callisto_catalina.'
              suffix='.log'
              timestamp='true'/>

      </Host>
    </Engine>
  </Service>
</Server>
```

As noted earlier, if a standalone configuration of Tomcat is being used, then only the HTTP/1.1 Connector is needed, and you are done with your configuration.

If you are running Tomcat with Apache, then you need the AJP/1.3 Connector, and have to update the Apache configuration, too. Because you are running two Tomcat instances on two different ports, each with an AJP 1.3 Connector, you need to modify the `workers.properties` file to point them to it, as shown:

```
# Define all the workers
worker.list=europa-worker, callisto-worker
# Set properties for callisto-worker
worker.callisto-worker.type=ajp13
worker.callisto-worker.host=callisto.dom
worker.callisto-worker.port=8109
```

(continued)

```
# Set properties for europa-worker
worker.callisto-worker.type=ajp13
worker.callisto-worker.host=europa.dom
worker.callisto-worker.port=8209
```

All that is required now is to start the two instances of Tomcat with the `CATALINA_BASE` set to the `catalina` subdirectory of the domains. To aid in the system administration task, you should write a shell script to start all the Tomcat instances as required. In this example configuration, create the following shell script with the name `start_sites.sh` in `CATALINA_HOME` and make it executable.

```
#!/bin/bash
CATALINA_HOME="/usr/local/tomcat"
SITE_ROOT="/home/websites"
SITES=`ls ${SITE_ROOT}`
for x in ${SITES}
do
        CATALINA_BASE=${SITE_ROOT}/${x}/catalina
        echo "Starting server: ${x} with CATALINA_BASE=${CATALINA_BASE}"
        export CATALINA_BASE
        ${CATALINA_HOME}/bin/startup.sh
done
```

Similarly, create a shell script for shutting down all the servers. This shell script is created with the name `shut_sites.sh` in `CATALINA_HOME` and should also be made executable:

```
#!/bin/bash
CATALINA_HOME="/usr/local/tomcat"
SITE_ROOT="/home/websites"
SITES=`ls ${SITE_ROOT}`
for x in ${SITES}
do
        CATALINA_BASE=${SITE_ROOT}/${x}/catalina
        echo "Shutting server: ${x} with CATALINA_BASE=${CATALINA_BASE}"
        export CATALINA_BASE
        ${CATALINA_HOME}/bin/shutdown.sh
done
```

Now, stop all the instances of Apache and Tomcat in the system. Start the Tomcat instances using the `start_sites.sh` script and follow it by starting Apache.

These startup and shutdown scripts can also be used in a system initialization script such as the `rc init` scripts kept in `/etc/init.d` on Linux systems. You can use these scripts as templates for your virtual hosting deployment.

Setting Memory Limits on the Tomcat JVM

Whether all the virtual hosts are running under the same Tomcat process or separate Tomcat processes are allocated for each of them, there is still the risk of a resource usage problem.

> *The problem is that a Java VM at startup allocates a fixed amount of memory for dynamic allocation. With several JVMs running, this number might be either too high (choking the virtual hosts that need more memory), or too low (causing suboptimal performance for the various hosts).*

Depending on the number and the type of virtual hosts running on the one machine, it is likely that a hosting environment will want to optimize these settings. This setting of memory (more specifically, heap memory that is used while allocating all dynamic data structures) is done by setting a command-line parameter for the Java executable when the Tomcat process is started.

The options that can be set are as follows:

- Initial Java heap size using the `-Xms` parameter
- Maximum Java heap size using the `-Xmx` parameter
- Java thread stack size using the `-Xss` parameter

For example, to set an initial heap size of 20MB (or 20 × 1024 × 1024 = 20971520), the value to be passed to the JVM as a parameter is `-Xms20971520` — (or the more succinct `-Xms20m`).

Factors Determining Memory Requirements

The nature of the applications being run on the JVM determines the optimum heap sizes. Heavy multi-threaded servers, such as Tomcat, that have a tendency to frequently allocate/de-allocate objects are quite sensitive to heap size because a lot of memory can be held up at times, waiting to be garbage-collected. Increasing the heap size in such scenarios can help a lot.

Conversely, very large heap sizes should be avoided by keeping `-Xmx` low. If other apps overload the machine, the heap could start using the swap space (that is, space allocated on the hard disk as an extension of RAM, when all the memory in RAM has been used up). This is also known as *virtual memory*. It reduces the performance of the system significantly. While a reasonable amount of swap usage is common for production servers, serious cases of continuous swapping, commonly known as *thrashing*, could slow the machine to a crawl.

The JVM normally starts with as little memory as possible, as specified in `-Xms`, and then slowly increases memory needs as required by the application, to the limit specified in `-Xmx`. If there is enough memory, it is possible to set `-Xms` to the same value as `-Xmx`. This can result in a faster startup time for the Java application. Always keep `-Xms` to a reasonable size to make applications more responsive.

The default value for `-Xms` and `-Xmx` (which differs from platform to platform) is normally too small for server applications. In addition, the heap resizing from `-Xms` to `-Xmx`, occurring slowly over time, causes the server to slowly pick up performance. To reduce this startup latency, you can even try setting these limits to the same value. However, if memory needs are minimal, this configuration will lose the advantage of the JVM automatically choosing the optimum heap size (between `-Xms` and `-Xmx`) for the Tomcat server.

When adding processors to a Symmetric MultiProcessing (SMP) machine, be sure to increase memory, because unlike memory allocations, which can be parallelized over the SMP, garbage collection cannot be. Therefore, it can soon become the bottleneck in the application.

Heavy, database-oriented applications consume a lot of memory because of result sets, temporary tables resulting from `JOIN` statements, and so on.

Ultimately, optimum heap sizes can be determined only by examining specific parameters such as the following:

- How many Tomcat instances will be running?
- What kind of traffic is expected by the site?

- ❑ Does the Web application use a lot of data transactions involving heavily filled up databases?
- ❑ How much RAM is available to be installed in the machine?
- ❑ How many processors does the machine have?

Setting Memory Limits in Tomcat

The parameters for setting the memory that is passed to the JVM can be set in the environment variable `JAVA_OPTS`. For easy system administration, the simplest option is to modify the multiple-Tomcat process starting script created earlier to send these options to the JVM so that each of the virtual hosts is restricted to these limits.

The modified script would look like the following. Here, the configuration parameters set the minimum and maximum heap limits of each Tomcat instance to 20MB and 50MB, respectively:

```
#!/bin/bash
JVM_OPTIONS='-Xms20m -Xmx50m'
export JAVA_OPTS
CATALINA_HOME="/usr/local/tomcat"
SITE_ROOT="/home/websites"
SITES=`ls ${SITE_ROOT}`
for x in ${SITES}
do
        CATALINA_BASE=${SITE_ROOT}/${x}/catalina
        echo "Starting server: ${x} with CATALINA_BASE=${CATALINA_BASE}"
        export CATALINA_BASE
        ${CATALINA_HOME}/bin/startup.sh
done
```

It is important to modify the values in `JVM_OPTIONS` as appropriate for the specific hosting requirements.

Using Java Security Manager Restrictions

The Java Security Manager allows you to set restrictions on what the Web application can do — which JAR files it can access, which network sockets it can open, which directories it can read/write to, and so on. By restricting what individual Web applications can do, you can protect each customer's Web applications and data from malicious or accidental access by other Web applications.

If you are running the virtual hosts in the same JVM, you can add restrictions per Web application, as shown:

```
#
# Give callisto.dom's web application specific permissions.
#
grant codeBase "file: /home/websites/callisto.dom/webapps/ROOT/WEB-INF/-" {
permission java.io.FilePermission "/some/callisto.dom/private_resource/*", "read";
 ...
};
```

If you are running separate JVMs for each virtual host as described in the section "Creating Separate JVMs for each Virtual Host," you can have a custom Java Security Manager policy for each JVM either by

passing the policy filename via the `-Djava.security.policy` parameter setting in `JAVA_OPTS`, or by putting the policy in each virtual host's configuration directory (i.e., at `CATALINA_BASE/conf/catalina.policy`).

Specific details about using a Java Security Manager are covered in Chapter 14.

Summary

This chapter covered various topics related to using Tomcat-based sites in a shared hosting scenario. The following are some of the key points discussed:

- Apache's HTTP server and Tomcat support IP as well as name-based virtual hosting.
- Tomcat can be set up for virtual hosting both in its standalone configuration as well as with an Apache HTTP server frontend.
- Running separate JVMs for each virtual host is a recommended security practice, although it is resource intensive.
- Controlling resource usage is important in a shared environment, and hence parameters such as initial/maximum heap size and thread stack size can be configured for each virtual host.

This chapter specifically discussed the configuration of Tomcat for a shared hosting scenario. More information on configuration of Apache HTTP Server and Tomcat for load balancing can be found in Chapter 11.

The next chapter covers how to monitor and manage Tomcat using Java Management Extensions (JMX).

16

Monitoring and Managing Tomcat with JMX

As a Tomcat administrator, you are likely to be completely comfortable when working with XML configuration files, examining detailed logs, and interacting with management applications such as the Tomcat Manager application. Logs provide debug traces when configuration mistakes are made, or when a system exception occurs. The Manager application enables remote start, stop, installation, and removal of applications. It also provides visibility into some information on the running applications. By and large, however, while a Tomcat server is up and running, there is very little you can do to ascertain the current state-of-health of Web applications and the Tomcat server. This is not really limited to the Tomcat server; there really is a lack of monitoring and management facilities for servers created on the Java platform — that is, until Java SE 5!

Starting in Java SE 5, and improved in Java SE 6, are the features known as Monitoring and Management (M and M). The M and M platform and tools support enable applications to be monitored and managed in real time — while they are running — either locally on the same machine or remotely over a network. The enabling technology is actually called Java Management Extension (JMX). Support for JMX is built extensively into the Tomcat server since version 4, and is relatively mature in Tomcat 6. This chapter discusses the following topics:

- ❏ JMX and manageability requirements
- ❏ Introduction to JMX
- ❏ JMX internals
- ❏ Configuring Tomcat 6 for JMX operations
- ❏ Using Manager Application JMX Proxy to monitor and manage Tomcat instances
- ❏ JDK tools for monitoring and management
- ❏ Utilizing the JMX support of Tomcat 6

A hands-on example shows you how JMX can be used to obtain live runtime information directly from the components that make up the Tomcat 6 server.

Administrators should be familiar with JMX and its implications because most administrative and management tools for Tomcat are based on JMX. Using standard tools included with Java SE 6 (JDK 1.6), an administrator can help you monitor the health, and manage the operation, of your Tomcat 6 servers.

The Requirement to Be Manageable

The original intent of Tomcat was to provide a workable reference implementation of the Servlet and JSP specifications. Because of its reliability, however, Tomcat has been increasingly adopted for production purposes. Features formerly offered only as enhancements by commercial vendors are making their way into the Tomcat wish list. Near the top among the list of requirements is a well-defined means of configuring, administrating, monitoring, and managing a large group of servers or server clusters.

Because of the increasing demand from today's Web applications and Web services, many Tomcat deployments span multiple servers. To provide for the scalability, availability, and throughput demands, many production environments involve multiple physical servers.

Added to this trend is the increasing popularity of shared hosting for JSPs, servlets, and Web applications. Shared hosting provides a cost-effective way to deploy Web applications for consumption over the Internet. In a typical hosting center, an individual physical server machine may have many independent JVMs concurrently running Tomcat. Often, a single instance of Tomcat may provide service for tens of virtual hosts. Until the arrival of JMX, managing these Tomcat and virtual host instances across a network of servers was an administrator's nightmare. Ad hoc solutions often require painstaking custom coding, are operating-system dependent, and are difficult to maintain. Figure 16-1 illustrates this management problem with Tomcat servers.

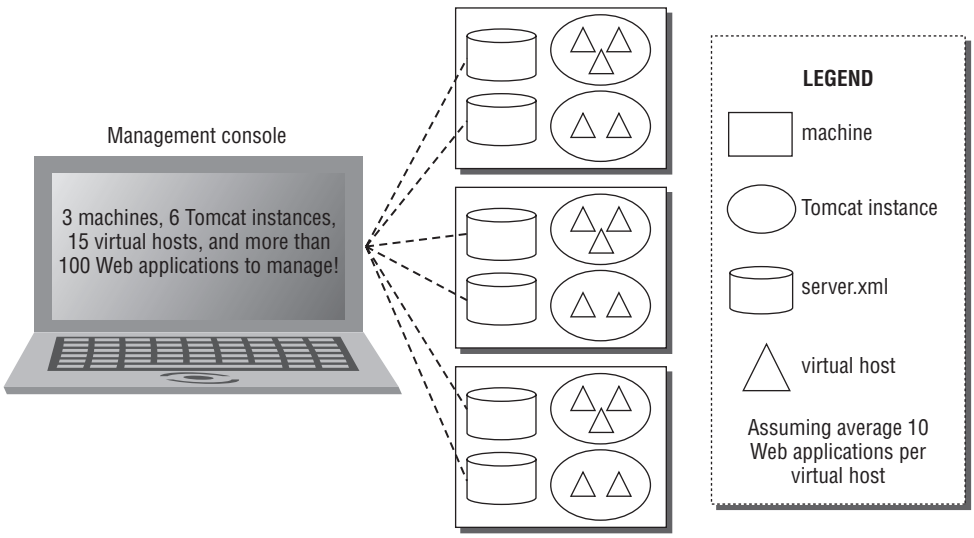

Figure 16-1: Management of a bank of Tomcat servers is problematic.

In Figure 16-1, multiple Tomcat instances are being managed from a single location. Multiple Tomcat instances are used frequently to implement Web application isolation, where you want to segregate applications with differing hardware and/or resource requirements. Isolation via separate JVM processes, even on the same machine, is useful when you don't want one crashing application to affect other running applications. Each instance can run multiple virtual hosts; virtual hosts enable shared hosting and running multiple domains on one physical machine (or Java VM). The management in Figure 16-1 involves ad hoc maintenance and editing of individual configuration files at each of the physical servers. Operating system–specific scripts are used to automate certain frequently repeated operations. These scripts are difficult to maintain and must be changed for servers with different operating systems.

Tomcat 6's JMX support can eliminate this situation. Management, monitoring, and administrative tools created using Java can access and control Tomcat servers (and components within them, such as virtual hosts) across a network with ease. All the benefits of the Java language (including write-once run-anywhere behavior and easy code maintainability) can be leveraged in all these new management solutions. Figure 16-2 illustrates the JMX solution to Tomcat server management.

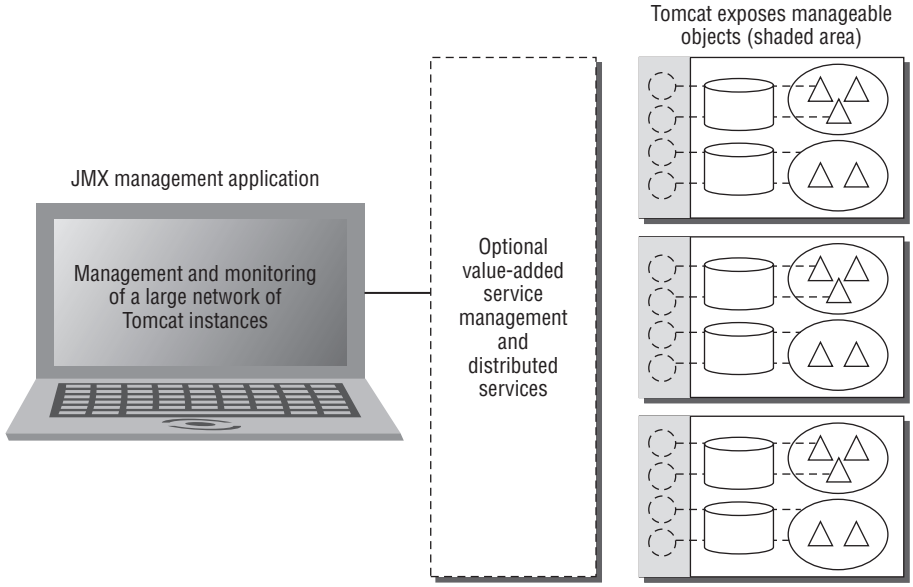
Tomcat exposes manageable objects (shaded area)
JMX management application
Management and monitoring of a large network of Tomcat instances
Optional value-added service management and distributed services

Figure 16-2: JMX solution to the Tomcat management problem

In Figure 16-2, Tomcat 6 is JMX-ready. Tomcat exposes a set of Java-based objects for external management. This set of objects involves all configurable aspects of Tomcat, and provides runtime control for operational tuning/tweaking and runtime statistics for monitoring. JMX enables these objects to be accessed by local or remote management agents. Optional distributed services and value-added agent logic may further consolidate and intelligently aggregate the exposed information, providing users with a simplified view of the management application.

This is a definite improvement over the ad hoc management alternative. The following section examines exactly what JMX offers.

All About JMX

JMX is the subject matter of JSR-3, a specification developed over several years by an industry-wide network management experts group that is part of the Java Community Process (JCP). Major vendors of network management systems have active representation in this group and have ensured a balance of coverage. The main goal of this group was to develop a set of well-defined and related specifications that together describe an architecture for the management of manageable entities (devices, software services, and so on) over a network using the Java programming language. The deliverables from this exercise include the following:

- A set of specifications with a detailed architectural framework, including detailed descriptions of required components and their operations
- A functional reference implementation, with associated documentation
- A compatibility test suite to test compliance with the specification

While the subject of network management or enterprise management is certainly not new, a Java-specific standard is. Until JMX, most network management systems (sometimes called enterprise-management systems) were proprietary in nature. They were typically designed using specific operating system and programming language combinations. This led to a proliferation of different versions of the same software base, maintained for different platforms.

The entities managed by a network management system have traditionally been hardware devices. Because of this origin, the de facto standard that makes devices generically manageable over a network is called *Simple Network Management Protocol (SNMP)*. The protocol was originally defined for hardware devices and is very restrictive when applied to modern software services that must be managed (such as Tomcat 6 servers). Because of the turbulent evolution of the network management industry, many proprietary extensions in addition to the SNMP protocol were introduced for vendor-specific device/management system combinations.

With the benefit of hindsight, JMX was designed to be flexible and adaptive from the start. Leveraging the benefits of the Java platform, the JMX specification is designed to facilitate the development of manageable entities and management systems without cornering vendors into restrictive implementations. At the same time, it must also be capable of coexisting and interoperating with the extensive body of already existing SNMP-based (or proprietary) managed systems in an evolutionary manner.

The JMX Architecture

Distilling decades of accumulated experience, the designers involved in JSR-3 made sure that the result is flexible and adaptable to both existing and future manageability needs. The architecture of JMX is layered as well as modular. The layers are well defined, yet loosely coupled. Existing networking and Java standards are used wherever applicable. As a result, the components of each layer can be built without prior intimate knowledge of the others. Being well defined and standards-based ensures that any specific implementation of one layer will directly work with an implementation of another.

Each layer within the JMX cake is called a *level*. The three levels in the JMX architecture are as follows:

- Instrumentation
- Agent
- Distributed services

Figure 16-3 shows a high-level view of the levels and components involved in a JMX-based system. The next section explores these levels.

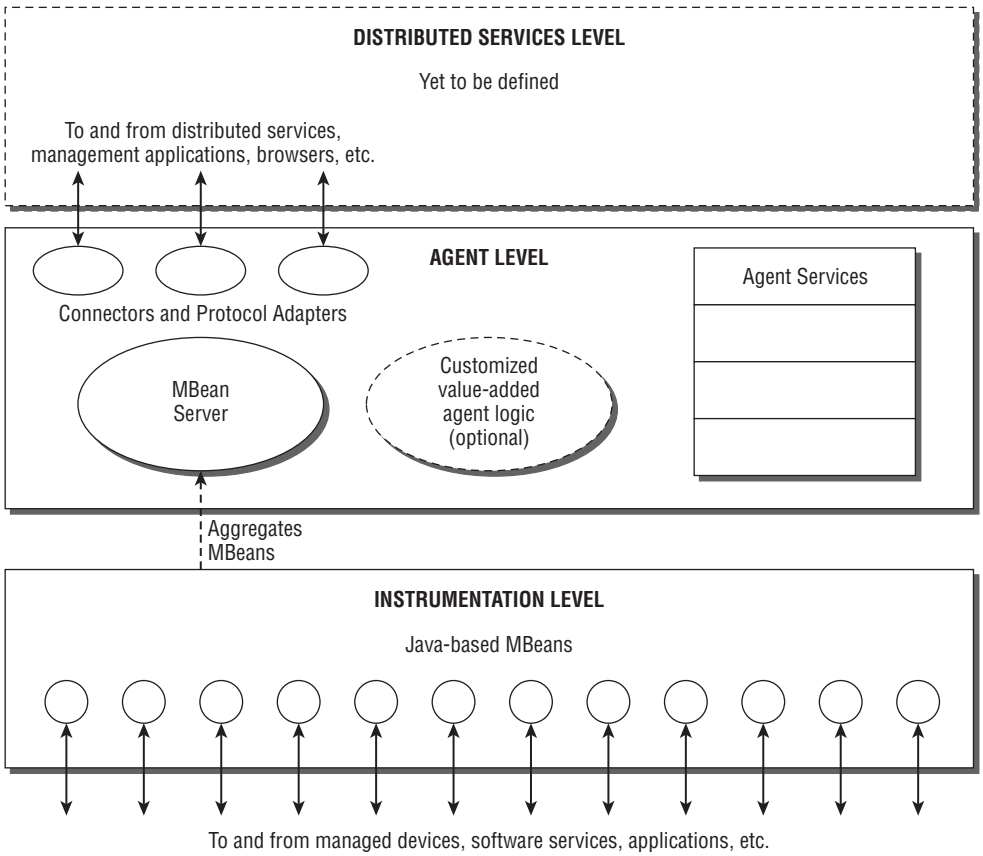

Figure 16-3: JMX architecture

The complete JMX picture involves many different related standard specifications (JSRs). In fact, each JMX level is defined in its own specification, created by a different group of experts. Taken together, there are quite a few JMX-related JSR specifications, and this number can grow as the different levels become more fully defined. Currently, the two most important specifications to Tomcat administrators are the following:

- Java Management Extension Instrumentation and Agent Specification 1.2
- Java Remote API Specification 1.0

This chapter summarizes the important aspects of JMX that are relevant to Tomcat administration. You should consult the specifications for other details.

Now let's take a look at the components and interactions within each of the JMX levels.

Instrumentation Level

Within the *instrumentation level*, resources and objects that can be managed and/or monitored are enumerated. The act of defining what can be managed/monitored is called *instrumentation*. For example, the set of Web applications running in a Tomcat 6 server may be a candidate for JMX management, and thus can be instrumented at the instrumentation level. This entails the creation of *Managed Beans (MBeans)*.

MBeans are Java components that live within the instrumentation level. They present a well-defined Java interface for the resources being managed. The resources, in this case, can be a hardware device, a software service, an application, or another entity (such as Web applications in a Tomcat 6 server). Instrumentation can be added to any existing resource simply by creating an MBean for the resource and hosting it on a network-connected Java VM. Typically, this is in the same physical box as the resource being managed. Figure 16-4 shows the high-level anatomy of an MBean.

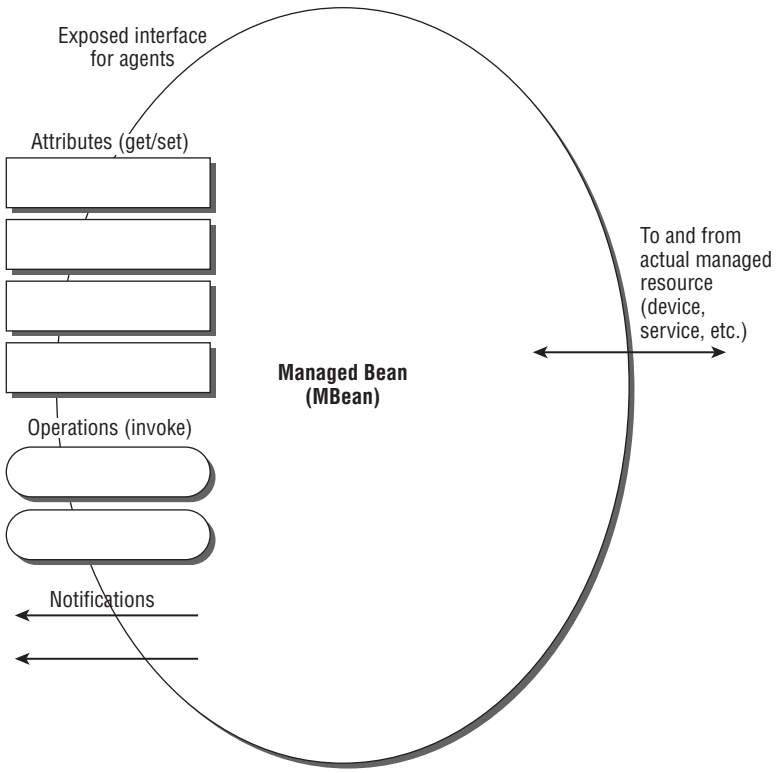

Figure 16-4: A Managed Bean (MBean)

Conceptually, MBeans are similar to JavaBeans, and expose attributes (such as properties on which you can use get and set), operations (such as methods that can be invoked), and notification events (distributed over the network, in this case) that can be caught by other levels. MBeans are oblivious to the agent that may be managing them, enabling instrumented resources to be managed by any JMX-compliant system.

Agent Level

At the agent level, the MBeans exposed by the instrumentation level are aggregated by the agent logic and presented as manageable entities to higher-level distributed services and/or management applications. Most agents simply provide the same interface exposed by the MBeans for management. However, the interface exposed to higher levels need not be the same as those exposed by the individual aggregated MBeans. For example, an intelligent agent may expose a single attribute called `GlobalMaxThreads` that will update attributes of threadpool management MBeans across all the Tomcat 6 server instances that the agent is aggregating.

Another major purpose of the agent level is to decouple the interdependency between the management applications and the resources being managed. This decoupling enables instrumentation to be added independently (by coding MBeans), without being concerned about how the MBeans will ultimately be used. For example, hundreds of MBeans have been created and are available within any running Tomcat 6 server today. However, JMX management applications that make clever use of these MBeans are yet to be created.

The agent level can mediate and facilitate multiple concurrent accesses to the set of aggregated MBeans. It can also convert to and from different access protocols, enabling access from different management applications/services. Other value-added features at the agent level may include monitoring, data filtering, data reduction, intelligent consolidation, and so on.

Figure 16-3 shows the components that reside in the agent level, including the following:

- **MBean Server:** Aggregates MBeans; receives runtime registration of MBeans and makes them available for external management via connectors and protocol adapters.
- **Connectors and protocol adapters:** Provides external access to the MBeans and services aggregated by the MBean Server.
- **Agent Services:** A mandatory standard set of services that are available to a developer creating MBeans or agent logic.

The following sections describe each of these components in more detail.

The MBean Server

The MBean Server is a component in the agent level that aggregates the MBeans being exposed by the agent. A managed device or service needs to register the MBeans that it wants to expose with the MBean Server before it can be accessed by higher-level management applications. This registration happens at runtime (when a JMX-managed box is booted, at the startup of the Tomcat 6 server, and so on).

Management applications can access MBean attributes, operations, or events only through the management interface provided by the MBean Server. The access is not direct, as most higher-level management applications are external to the process/physical box of the server/device being managed. Instead, it is performed through the assistance of connectors and protocol adapters (described in the next section).

Note that many agent-level services are typically also implemented as MBeans. In these cases, they must also be registered with the MBean Server.

Connectors and Protocol Adapters

Management applications can access managed resources only through the MBean Server component at the agent level. However, the MBean Server component has no direct means of communicating with external applications. Instead, the MBean Server relies on available connectors and protocol adapters to interface with external management applications, as shown in Figure 16-5.

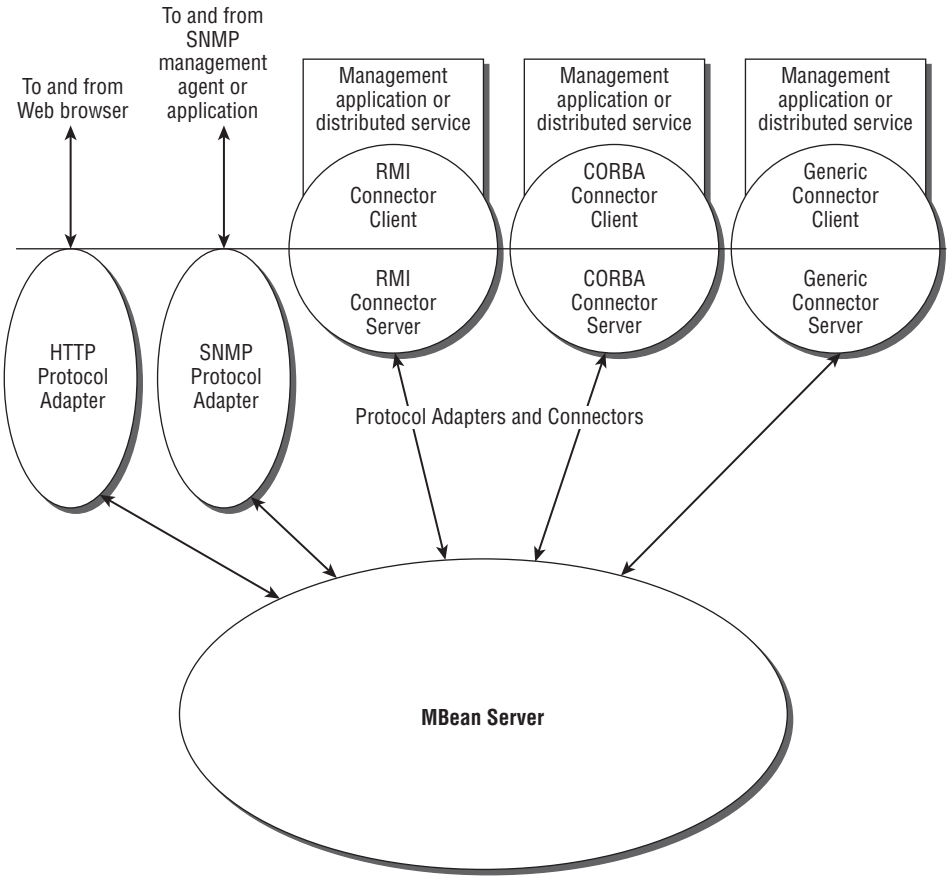

Figure 16-5: Connectors and protocol adapters

Connectors typically utilize standard remote-access mechanisms (RMI, CORBA, Java Socket, and so on) to directly expose the interface of the agent remotely. Protocol adapters provide access to the agent by mapping JMX requests to and from specific protocols. For example, an HTTP protocol adapter may enable users to directly access the agent using a simple browser.

The JMX agent logic within Tomcat 6 supports both an HTTP protocol adapter in the form of the Manager application's JMX proxy (more details about this later in this chapter), and an RMI connector for remote JMX access. Figure 16-6 shows the connector and protocol adapter support within Tomcat 6.

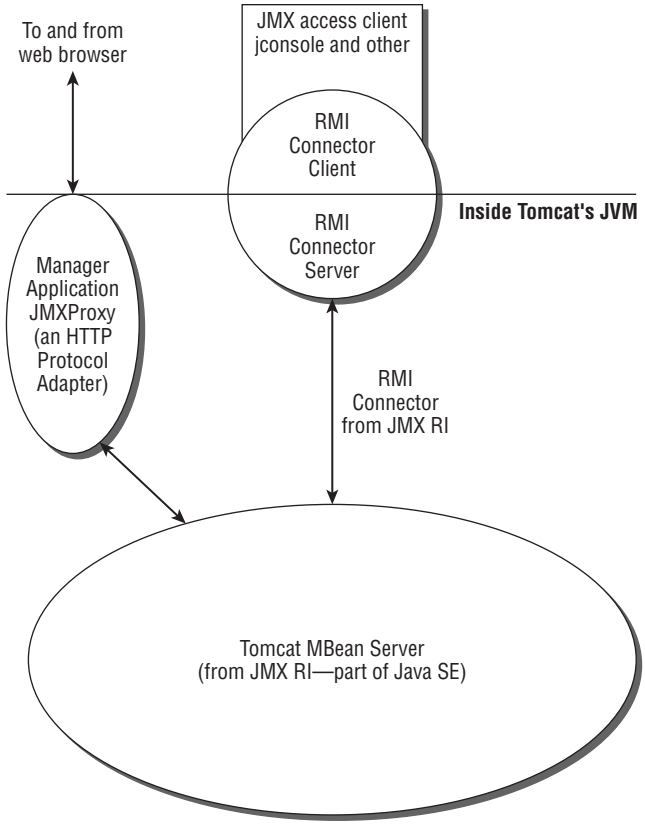

Figure 16-6: Tomcat 6 agent implementation

Agent Services

JMX specifies a set of mandatory services that every compliant agent implementation must provide. These services include the following:

- A dynamic class loading service for loading management applets (mlets)
- A monitoring service to monitor changes in specified MBean attributes
- A timer service for periodic or one-shot timing
- A relation service for enforcing relationships between MBeans

Agent services are used mainly by developers to implement customized agent logic and/or management applications. Tomcat administrators need not be concerned with their programmatic features. However, an awareness of their existence may facilitate useful discussions with development and network management staff.

Distributed Services Level

The *distributed services level* is the topmost level in the JMX architectural diagram (see Figure 16-3). This level includes applications and services that access the functionality provided by the agent level.

This level specifies standard high-level management interfaces for the creation of management applications. These interfaces and their associated components enable management applications (or other services applications) to interact with the agent level. The specification for the distributed services level is still a work in progress and has not yet been finalized. No further public information is currently available.

JMX Remote API

While connectors and protocol adapters can be seen as agent level components, specifications for them are not part of JMX Instrumentation and Agent 1.4 specification. Instead, different connectors and adapters are covered under different specifications. Specifically, JSR-160 (the JMX Remote API specification) covers the details for a set of connectors that can be used to remotely access the agent level.

Originally, it was called JMX Remoting, and was destined to be included with a new release of the base JMX. However, the experts group decided to factor out elements that enable remote access to the agent layer and place them in their own specification. This enabled the earlier release of the base specification.

Before working with Tomcat 6's JMX agent, you should have an understanding of how developers typically create MBeans. This is necessary because the terminology introduced is a prerequisite for the Tomcat 6 MBeans examined later in this chapter.

An Anthology of MBeans

MBeans are software modules that expose the capabilities of a hardware device, a software service, or a software component. In JMX literature and technical discussions, you will see references to different types of MBeans, including the following:

- Standard MBean
- Dynamic MBean
- Model MBean
- Open MBean

It is not necessary for administrators to fully understand their differences. They all appear the same to management software that may work with them. Developers, however, will be very intimate with their differences. The following sections provide a brief explanation of each.

Standard MBeans

In standard MBeans, the features (attributes, operations, and notifications) that are exposed for management are fixed and cannot change (without a software or firmware change). Standard MBeans are typically the easiest to code for developers.

Dynamic MBeans

In dynamic MBeans, the features that are exposed for management are determined at runtime. Therefore, the exposed features can change over time.

Model MBeans

Because of the flexible nature of dynamic MBeans, they are quite difficult and tedious to code. Model MBeans are a type of dynamic MBean used by developers to expedite the creation of their own dynamic MBeans. For example, Tomcat 6 makes extensive use of the model MBean support provided by the Apache Jakarta Commons (modeler) project.

Open MBeans

Open MBeans are a type of dynamic MBean. The unique quality of open MBeans is their capability to be managed by any compliant management software. The exposed features of an open MBean are guaranteed to be compliant with a universally manageable set of data types. Management software can then explore the set of exposed features at runtime to use the MBean. Open MBeans are a new feature of JMX 1.2, and are not required for compliance with previous JMX versions.

With all the basic technicalities out of the way, it's now time to take a look at the manageable elements exposed by Tomcat 6 through JMX MBeans.

JMX Manageable Elements in Tomcat 6

Almost all configurable elements of Tomcat 6 will become JMX-manageable in the near future. As developers build new attributes into the Tomcat 6 architectural components, they will create the corresponding MBean to expose these attributes for external management and monitoring.

In addition, the Tomcat 6 server creates some MBeans at runtime (see Chapter 6) to make dynamic runtime-only objects manageable. For example, elements in a UserDatabase (such as Users, Roles, and Groups) are created at runtime. When a Web application is deployed or redeployed, Tomcat 6 also creates a manageable object for the context that supports the application.

Much of this support is still a work in progress with Tomcat 6 (to be solidified in later releases). The following list describes MBean-exposed objects in Tomcat 6 and the attributes that are accessible via JMX:

- **Manageable Tomcat 6 architectural components**: These include `Service`, `Server`, `Engine`, `Connector`, and `Host`.
- **Manageable nested component**: These include `Realm`, `Logger`, `Valve`, and `Manager`.
- **Manageable runtime data objects**: These include `UserDatabase`, `User`, and `Role`.
- **Manageable resource objects**: These include `NamingResources`, `Environment`, `Resource`, `ResourceLink`, application-related objects, `WebModule`, internal Tomcat objects, `RequestProcessor`, `Cache`, and `ThreadPool`.

Included with the following discussion of these items are step-by-step instructions for accessing these manageable components via either of the following means:

- The JMX proxy of the Manager application (this is, architecturally, an HTTP protocol adapter)
- The RMI connector

The listing can be used as your reference guide to probe around the manageable components of Tomcat 6.

If you cannot wait to start probing around your own Tomcat 6 instance using JMX, skip right ahead to the section "Accessing Tomcat 6's JMX Support via the Manager Proxy," later in this chapter and consult the MBean list as necessary for reference.

Manageable Tomcat 6 Architectural Components

The first sets of manageable objects are the components that you are familiar with when working with the `server.xml` file or the `admin` application. This section describes their accessible attributes, together with the data type expected for the attribute and whether the attribute is read-only (RO) or read/write (R/W).

If you need a refresher on any of the manageable components and/or their configurable attributes, you may want to revisit Chapters 5 and 6.

Service

Note that the `modelerType` attribute exists on every manageable component because Tomcat 6 uses library code from the Apache Jakarta Project's Commons Modeler to simplify its own implementation of JMX. MBeans created using Apache's Commons Modeler typically expose this `modelerType` attribute.

In the following table, the column labeled Read/Write indicates whether an attribute can be changed (R/W) or cannot be changed (RO — Read-Only) via JMX control.

Attribute Name	Read/Write	Type
`modelerType`	R/W	`java.lang.String`
`managedResource`	R/W	`java.lang.Object`
`name`	R/W	`java.lang.String`
`connectorNames`	RO (**R**ead **O**nly)	`javax.management.ObjectName[]`
`containerName`	RO	`java.lang.String`
`container`	RO	`javax.management.ObjectName`

Server

Because JMX MBeans expose both configuration and runtime attributes, many new attributes provide a runtime relationship between the components. For example, the `serviceNames` attribute provides an array of services associated with the Server component, as shown in the following table.

Attribute Name	Read/Write	Type
`modelerType`	R/W	`java.lang.String`
`managedResource`	R/W	`java.lang.Object`
`port`	R/W	`int`
`shutdown`	R/W	`java.lang.String`
`serviceNames`	R/W	`javax.management.ObjectName[]`

Engine

In the following table, it is interesting to note that at runtime, the Engine maintains a list of configured Valve components. This list can be accessed via JMX using the `valveObjectNames` attribute.

Attribute Name	Read/Write	Type
`baseDir`	R/W	`java.lang.String`
`defaultHost`	R/W	`java.lang.String`
`jvmRoute`	R/W	`java.lang.String`
`managedResource`	R/W	`java.lang.Object`
`modelerType`	R/W	`java.lang.String`
`name`	R/W	`java.lang.String`
`realm`	R/W	`java.lang.String`
`valveObjectNames`	RO	`javax.management.ObjectName[]`

Connector

As shown in the following table, each connector configured has its own runtime MBean. Some connectors (for example, JK Connectors) may have additional runtime MBeans that are accessible via JMX.

Attribute Name	Read/Write	Type
`modelerType`	R/W	`java.lang.String`
`acceptCount`	R/W	`int`
`Address`	R/W	`java.lang.String`
`algorithm`	R/W	`java.lang.String`
`allowTrace`	R/W	`boolean`
`bufferSize`	R/W	`int`
`className`	RO	`java.lang.String`
`clientAuth`	R/W	`boolean`
`ciphers`	R/W	`java.lang.String`
`compression`	R/W	`java.lang.String`
`connectionLinger`	R/W	`int`
`connectionTimeout`	R/W	`int`
`connectionUploadTimeout`	R/W	`int`
`disableUploadTimeout`	R/W	`boolean`
`emptySessionPath`	R/W	`java.lang.String`
`enableLookups`	R/W	`boolean`

Table continued on following page

Attribute Name	Read/Write	Type
keepAliveTimeout	R/W	int
keystoreFile	R/W	java.lang.String
keystorePass	R/W	java.lang.String
keystoreType	R/W	java.lang.String
keyAlias	R/W	java.lang.String
maxHttpHeaderSize	R/W	int
maxKeepAliveRequests	R/W	int
maxPostSize	R/W	int
maxSpareThreads	R/W	int
maxThreads	R/W	int
minSpareThreads	R/W	int
port	R/W	int
protocol	R/W	java.lang.String
protocolHandlerClassName	RO	java.lang.String
proxyName	R/W	java.lang.String
proxyPort	R/W	int
redirectPort	R/W	int
scheme	R/W	java.lang.String
secret	WO	java.lang.String
secure	R/W	boolean
sslProtocol	R/W	java.lang.String
sslProtocols	R/W	java.lang.String
strategy	R/W	java.lang.String
tcpNoDelay	R/W	boolean
threadPriority	R/W	java.lang.String
tomcatAuthentication	R/W	boolean
trustStoreFile	R/W	java.lang.String
trustStorePass	R/W	java.lang.String
trustStoreType	R/W	java.lang.String
URIEncoding	R/W	java.lang.String
xpoweredBy	R/W	boolean

Host

As shown in the following table, each configured virtual host in the Engine exposes its own JMX-accessible MBean.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
appBase	R/W	java.lang.String
autoDeploy	R/W	boolean
debug	R/W	int
deployOnStartup	R/W	boolean
deployXML	R/W	boolean
managedResource	R/W	java.lang.Object
name	R/W	java.lang.String
realm	R/W	Org.apache.catalina.Realm
configClass	R/W	java.lang.String
unpackWARs	R/W	boolean
xmlNamespaceAware	R/W	boolean
xmlValidation	R/W	boolean
children	R/W	javax.management.ObjectName[]
aliases	R/W	java.lang.String[]
valveNames	R/W	java.lang.String[]
valveObjectNames	R/W	javax.management.ObjectName[]

Manageable Nested Components

Following is the set of noncontainer nested components that are exposed on Tomcat 6 via JMX. Chapter 6 provides a detailed description of most of their attributes. The existence and number of these elements depend on the current configuration (that is, `server.xml` settings).

Realm

The components in the following table correspond to a configured `<Realm>` element.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
className	RO	java.lang.String
resourceName	R/W	java.lang.String

Valve

The components in the following table correspond to a configured `<Valve>` element. Valves, because of their diverse application and implementation, can expose additional attributes.

Attribute Name	Read/Write	Type
`modelerType`	R/W	`java.lang.String`
`className`	RO	`java.lang.String`

Manager

The components in the following table correspond to a configured `<Manager>` element, representing a session manager implementation (that is, a Persistent Session Manager).

Attribute Name	Read/Write	Type
`modelerType`	R/W	`java.lang.String`
`algorithm`	R/W	`java.lang.String`
`randomFile`	R/W	`java.lang.String`
`className`	RO	`java.lang.String`
`distributable`	R/W	`boolean`
`entropy`	R/W	`java.lang.String`
`managedResource`	R/W	`java.lang.Object`
`maxActiveSessions`	R/W	`int`
`maxInactiveInterval`	R/W	`int`
`sessionMaxKeepAliveTime`	R/W	`Int`
`sessionAverageAliveTime`	R/W	`Int`
`processExpiresFrequency`	R/W	`int`
`name`	RO	`java.lang.String`
`pathname`	R/W	`java.lang.String`
`activeSessions`	RO	`int`
`sessionCounter`	R/W	`int`
`maxActive`	R/W	`int`
`rejectedSessions`	R/W	`int`
`expiredSessions`	R/W	`int`
`processingTime`	R/W	`long`
`duplicates`	R/W	`int`

Manageable Runtime Data Objects

Chapter 6's overview of lifecycle listeners reveals that Tomcat 6 uses lifecycle listeners to create MBeans for runtime objects. For example, the UserDatabase that is used by Tomcat for authentication and authorization (if it is configured to use a UserDatabase Realm) can be accessed via these JMX MBeans. The listeners also create MBeans for each user, role, and group within the database.

The following tables list the attributes that are available with these runtime data objects.

UserDatabase

In the following table, note that the UserDatabase contains attributes that contain users, groups, and roles information. This information is managed in memory by the UserDatabase Realm. UserDatabase is located under the `Users` key (separate from the main `Catalina` key) within jconsole.

Attribute Name	Read/Write	Type
`modelerType`	R/W	`java.lang.String`
`encoding`	R/W	`java.lang.String`
`groups`	RO	`java.lang.String[]`
`pathname`	R/W	`java.lang.String`
`roles`	RO	`java.lang.String[]`
`readonly`	RO	`boolean`
`writable`	RO	`booelan`
`users`	RO	`java.lang.String[]`

User

In the following table, note the wide-open availability of a password for the user via this MBean. This underscores the need for careful security considerations before enabling JMX access on production systems. The user is exposed under the `Users` key (separate from the main `Catalina` key) within jconsole.

Attribute Name	Read/Write	Type
`modelerType`	R/W	`java.lang.String`
`fullName`	R/W	`java.lang.String`
`groups`	R/W	`java.lang.String[]`
`password`	R/W	`java.lang.String`
`roles`	RO	`java.lang.String[]`
`username`	R/W	`java.lang.String`

Role

The following table lists the attributes available with the `Role` object.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
description	R/W	java.lang.String
rolename	R/W	java.lang.String

Manageable Resource Object

Any global resources, Web-application resources, or environments that are configured for JNDI access within the configuration files become manageable through an MBean instance.

NamingResources

As shown in the following table, this object maintains a live list of all the defined JNDI-accessible global naming resources.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
environments	RO	java.lang.String[]
resources	RO	java.lang.String[]
resourceLinks	RO	java.lang.String[]

Environment

As shown in the following table, any global <Environment> definition has an associated Environment MBean.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
className	RO	java.lang.String
description	R/W	java.lang.String
name	R/W	java.lang.String
override	R/W	boolean
type	R/W	java.lang.String
value	R/W	java.lang.String

Resource

As shown in the following table, any global <Resource> definition will cause the creation of an associated MBean instance with the following attributes.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
auth	R/W	java.lang.String
description	R/W	java.lang.String
name	R/W	java.lang.String
scope	R/W	java.lang.String
type	R/W	java.lang.String

ResourceLink

A <resource-link> definition creates an associated ResourceLink MBean instance with the attributes shown in the table that follows.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
global	R/W	java.lang.String
name	R/W	java.lang.String
type	R/W	java.lang.String

Exposed Application-Related Objects

Several internal Tomcat 6 objects correspond to J2EE-defined runtime objects. They are accessible via JMX and are described in the following sections.

WebModule

A WebModule is a deployable unit. Tomcat 6 keeps track of currently deployed modules internally, and the information can be accessed via the MBeans shown in the following table. A WebModule MBean roughly corresponds to the combination of a single context descriptor and a deployment descriptor.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
antiJARLocking	R/W	boolean
antiResourceLocking	R/W	boolean
allowLinking	R/W	boolean
annotationProcessor	R/W	org.apache.AnnotationProcessor
cacheMaxSize	R/W	int
cacheTTL	R/W	int
cachingAllowed	R/W	boolean

Table continued on following page

Attribute Name	Read/Write	Type
caseSensitive	R/W	boolean
children	R/W	javax.management.ObjectName[]
cookies	R/W	boolean
compilerClasspath	R/W	java.lang.String
configFile	R/W	java.lang.String
crossContext	R/W	boolean
defaultContextXml	R/W	java.lang.String
defaultWebXml	R/W	java.lang.String
delegate	R/W	boolean
deploymentDescriptor	R/W	java.lang.String
docBase	R/W	java.lang.String
engineName	R/W	java.lang.String
environments	RO	java.lang.String[]
eventProvider	R/W	boolean
javaVMs	R/W	java.lang.String[]
loader	R/W	org.apache.catalina.Loader
managedResource	R/W	java.lang.Object
manager	R/W	org.apache.catalina.Manager
managerChecksFrequency	R/W	int
mappingObject	R/W	java.lang.Object
namingContextListener	R/W	Org.apache.catalina.core .NamingContextListener
objectName	R/W	java.lang.String
override	R/W	boolean
parentClassLoader	R/W	java.lang.ClassLoader
path	R/W	java.lang.String
privileged	R/W	boolean
processingTime	RO	long
realm	R/W	org.apache.catalina.Realm
reloadable	R/W	boolean
resourceNames	RO	java.lang.String[]
saveConfig	R/W	boolean
server	R/W	java.lang.String
servlets	RO	java.lang.String[]
startTime	R/W	long

Attribute Name	Read/Write	Type
startupTime	R/W	long
state	R/W	int
stateManageable	R/W	boolean
statisticsProvider	R/W	boolean
staticResources	RO	javax.naming.directory.DirContext[]
swallowOutput	R/W	boolean
tldScanTime	R/W	long
unloadDelay	R/W	long
useNaming	R/W	boolean
valveObjectNames	RO	javax.management.ObjectName[]
welcomeFiles	RO	java.lang.String[]
workDir	R/W	java.lang.String

Servlet

As shown in the following table, each activated servlet also has its own MBean, which provides valuable information on the Web application and servlet granularity.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
engineName	RO	java.lang.String
eventProvider	R/W	boolean
objectName	R/W	java.lang.String
stateManageable	R/W	boolean
statisticsProvider	R/W	boolean
processingTime	RO	long
maxTime	RO	long
requestCount	RO	int
errorCount	RO	int
loadTime	RO	long
classLoadTime	RO	int

Exposed Internal Tomcat Objects

A few core internal Tomcat objects have MBeans exposed. The following sections describe several of the more interesting ones.

RequestProcessor

Each virtual host implements its own request processor. The following table shows how MBeans expose some interesting information. Information such as request count and error count may be relevant to administrators when troubleshooting.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
virtualHost	RO	java.lang.String
bytesSent	R/W	long
method	RO	java.lang.String
remoteAddr	RO	java.lang.String
requestBytesSent	RO	long
contentLength	RO	int
bytesReceived	R/W	long
requestProcessingTime	RO	long
globalProcessor	R/W	org.apache.coyote.RequestGroupInfo
protocol	RO	java.lang.String
currentQueryString	RO	java.lang.String
maxRequestUri	R/W	java.lang.String
requestBytesReceived	RO	long
serverPort	RO	int
stage	R/W	int
requestCount	R/W	int
maxTime	R/W	long
processingTime	R/W	long
currentUri	RO	java.lang.String
errorCount	R/W	int

Cache

As shown in the following table, the object cache and threadpool objects exposed through JMX can provide administrators with valuable information for tuning the performance of a Tomcat system or cluster.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
accessCount	RO	long
cacheMaxSize	R/W	int
hitsCount	RO	long

Attribute Name	Read/Write	Type
maxAllocateIterations	R/W	int
spareNotFoundEntries	R/W	int
cacheSize	RO	int
desiredEntryAccessRatio	R/W	long

ThreadPool

The following table shows attributes associated with the `ThreadPool` object.

Attribute Name	Read/Write	Type
modelerType	R/W	java.lang.String
name	R/W	java.lang.String
minSpareThreads	R/W	int
currentThreadsBusy	RO	int
daemon	R/W	boolean
threadStatus	RO	java.lang.String[]
sequence	RO	int
currentThreadCount	RO	int
maxSpareThreads	R/W	int
maxThreads	R/W	int
threadParam	RO	java.lang.String[]

Accessing Tomcat 6's JMX Support via the Manager Proxy

The Manager application (featured in Chapter 8) has a JMX proxy that can be used to interact with Tomcat's agent level directly. The proxy enables the monitoring of Tomcat components through the exposed MBeans. It also enables you to read the value of an MBean attribute, or change/set the value of writable MBean attributes. Figure 16-7 illustrates the operation of the JMX proxy.

In Figure 16-7, notice that the Manager application provides an HTML-based interface to the JMX MBean server, acting as an HTTP protocol adapter for the agent. The Manager application adds essential value in this scenario. It provides querying capabilities and will authenticate the user before granting access to the JMX proxy.

Architecturally, because the `manager` JMX proxy actually runs within the same JVM as the Tomcat server, it can be viewed as a part of Tomcat's agent-level implementation. As mentioned previously, it acts as an HTTP protocol adapter.

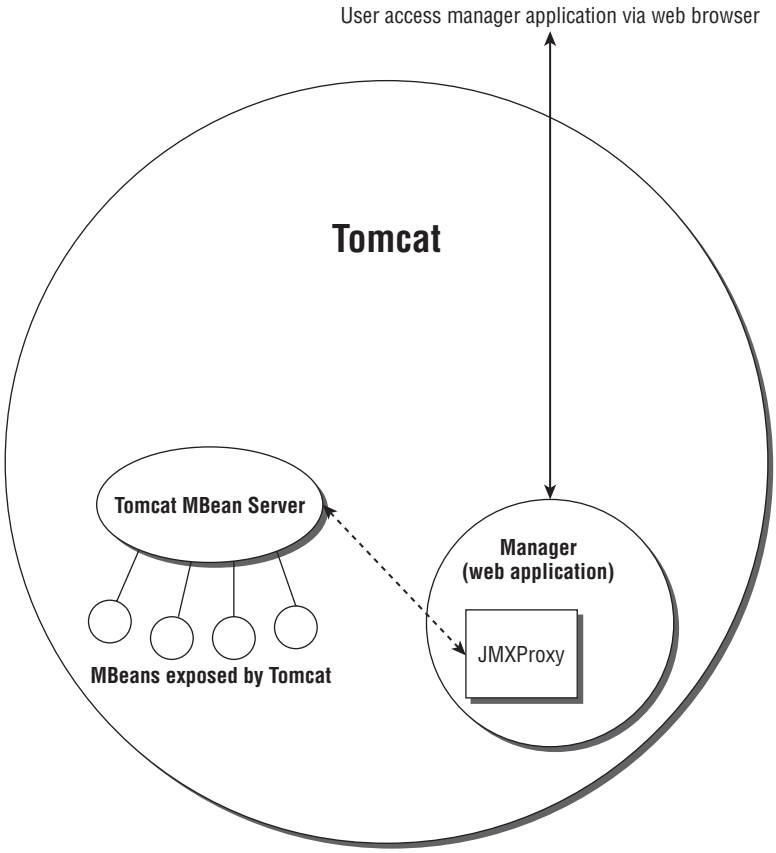

Figure 16-7: The Manager application's JMX proxy

Working with the JMX Proxy

The URL for accessing the JMX proxy using a browser is as follows:

```
http://<host>:<port>/manager/jmxproxy/<operation details>
```

If Tomcat is running locally with the default configuration, the URL is as follows:

```
http://localhost:8080/manager/jmxproxy/<operation details>
```

No stylized HTML Web pages or tables are displayed by this proxy. However, it is capable of performing the following operations against the Tomcat 6 agent (MBean Server) implementation:

- Query for MBeans and current attribute values
- Set MBean attribute values

WARNING: *Modifying the value of a Tomcat internal MBean during production operation can potentially cause problems that may result in an application and/or system crash. Use this feature at your own risk.*

The general form for a query operation using the JMX proxy is as follows:

```
?qry=<query details>
```

For example, you can get a complete listing of all the available MBeans using the following wildcard query:

```
http://localhost:8080/manager/jmxproxy/?qry=*:*
```

You must enable access to the Tomcat Manager application before this will work. To enable access to the Tomcat Manager, perform the following steps:

1. Shut down the Tomcat server if it is running.
2. Change the directory to `$CATALINA_HOME/conf`.
3. Open the `tomcat_users.xml` file with a text editor and modify the line with the `tomcat` user:

   ```
   <user username="tomcat" password="tompass" roles="manager,admin"/>
   ```

4. Start the Tomcat server.

When prompted for a user and password, you can use `tomcat` and `tompass` respectively.

The result will be similar to what is shown in Figure 16-8.

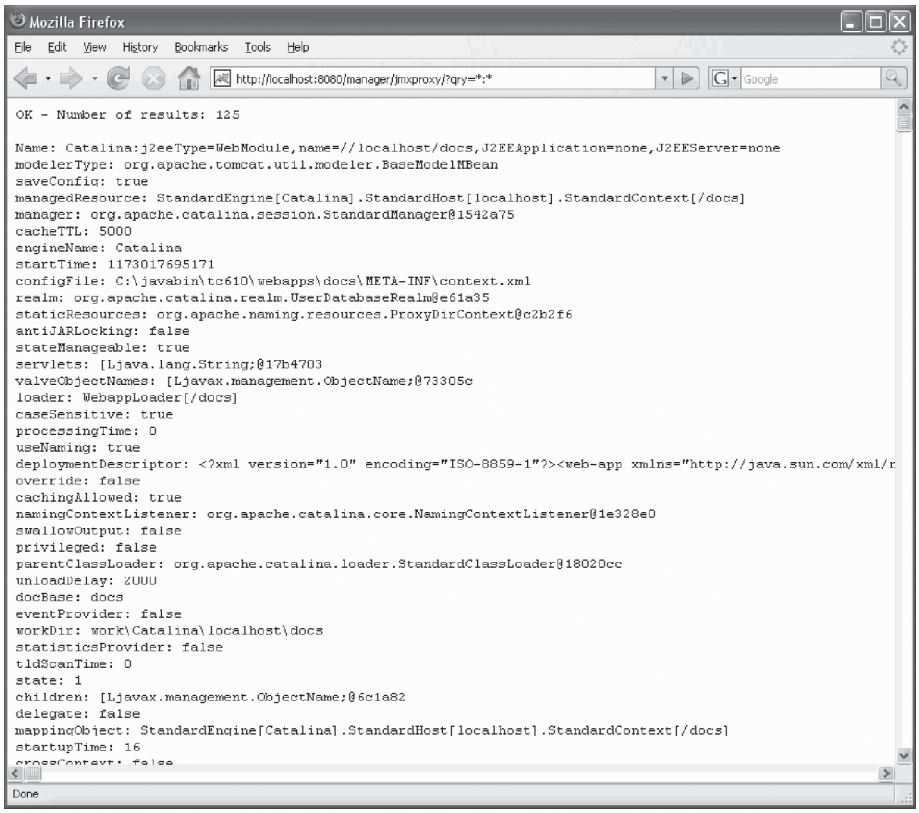

Figure 16-8: Listing of all the MBeans in Tomcat 6

As another example, you can get a listing of all the connector MBeans using the following query:

```
http://<host>:<port>/manager/jmxproxy/?qry=*:type=Connector,*
```

The result will be similar to what is shown in Figure 16-9.

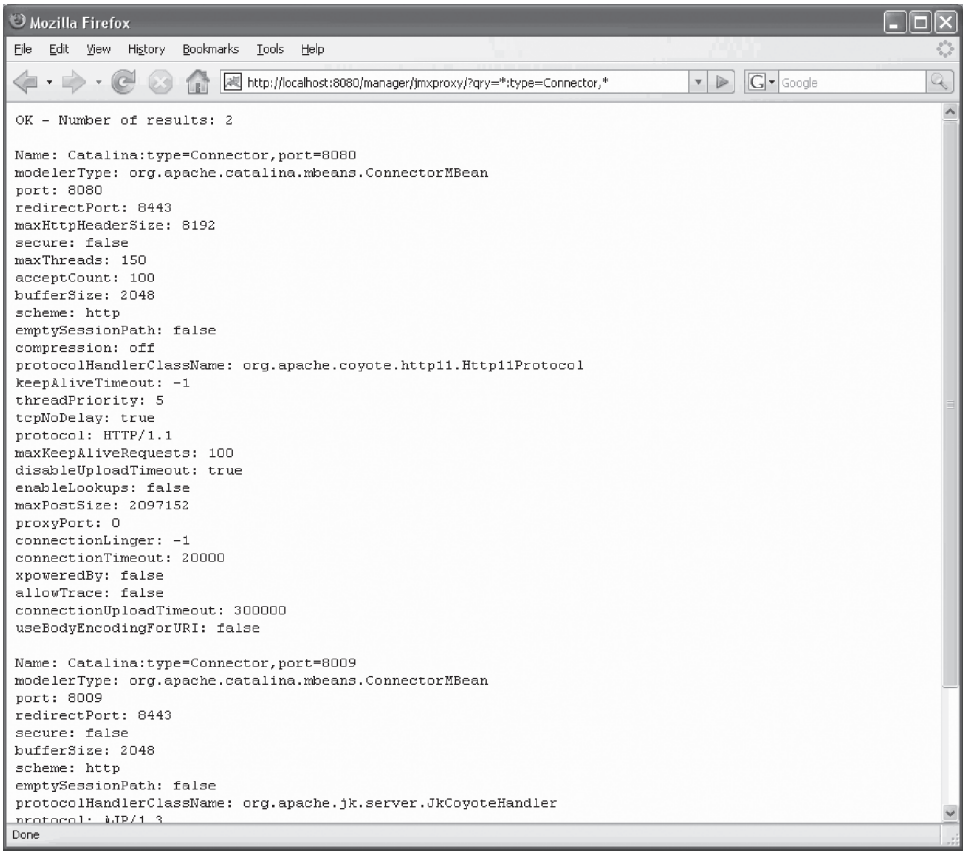

Figure 16-9: Listing of all Connector MBeans

Modifying MBean Attributes

Another operation you can perform with the JMX proxy is to change the attribute of an MBean. The general syntax for this operation is as follows:

```
http://<host>:<port>/manager/jmxproxy/?set=<full MBean name>&att=<attribute
name>&val=<value to change to>
```

For example, the following procedure can be used during performance tuning to change the maximum number of threads managed by Tomcat 6's threadpool in real time. You first need to query for the full MBean name of the exposed threadpool, as shown in the following example:

```
http://<host>:<port>/manager/jmxproxy/?qry=*:type=ThreadPool,*
```

The output of the query will be similar to what is shown in Figure 16-10.

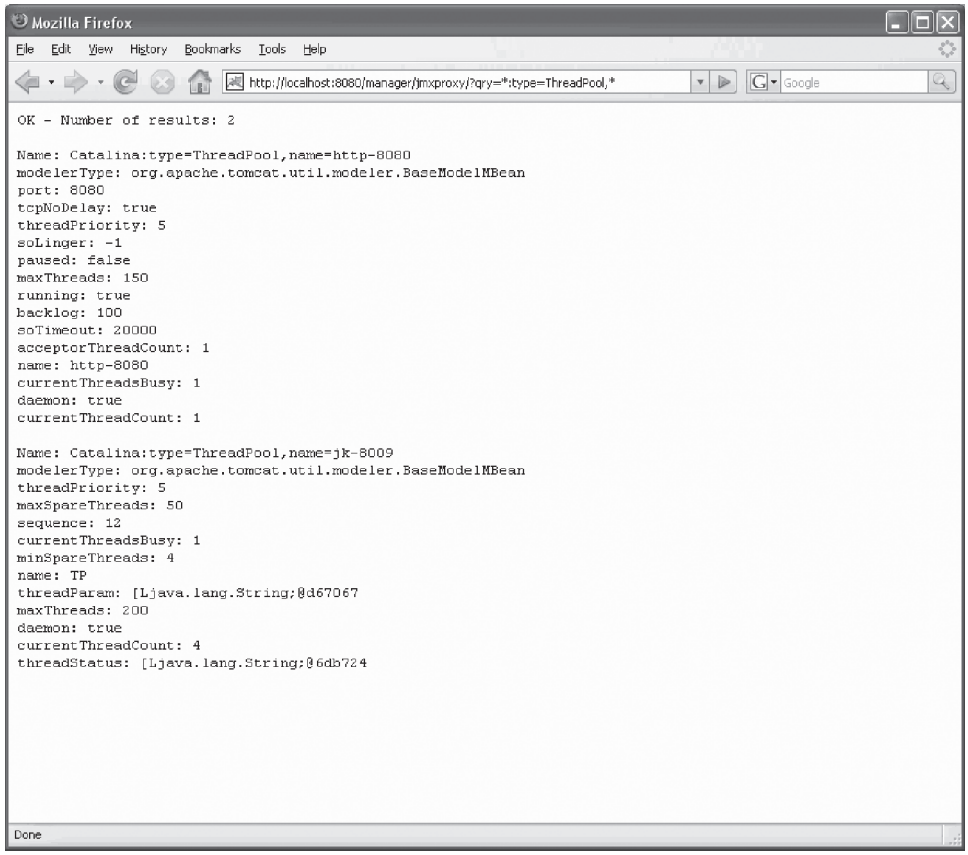

Figure 16-10: Querying for Tomcat 6's ThreadPool MBean name

The current value of the `maxThreads` attribute for the first `ThreadPool` is 150. It will be changed to 200.

Now, the full name of the MBean for this first threadpool is visible from the query output as follows:

```
Catalina:type=ThreadPool,name=http-8080
```

Next, by consulting the Tomcat 6 MBeans description listing in this chapter, you can determine that the `maxThreads` attribute is a writable property.

Finally, the URL for the `set` operation is as follows (type the entire URL on one line):

```
http://<host>:<port>/manager/jmxproxy/?set= Catalina:
type=ThreadPool,name=http-8080&att=maxThreads&val=200
```

Upon successful operation, the output is similar to what is shown in Figure 16-11. If you try to modify a read-only attribute, you will receive a message reporting an "attribute not found" exception.

Figure 16-11: Increasing the maxThreads attribute of Tomcat 6's ThreadPool object

If you perform another query for the ThreadPool information, you will see the change, as shown in Figure 16-12.

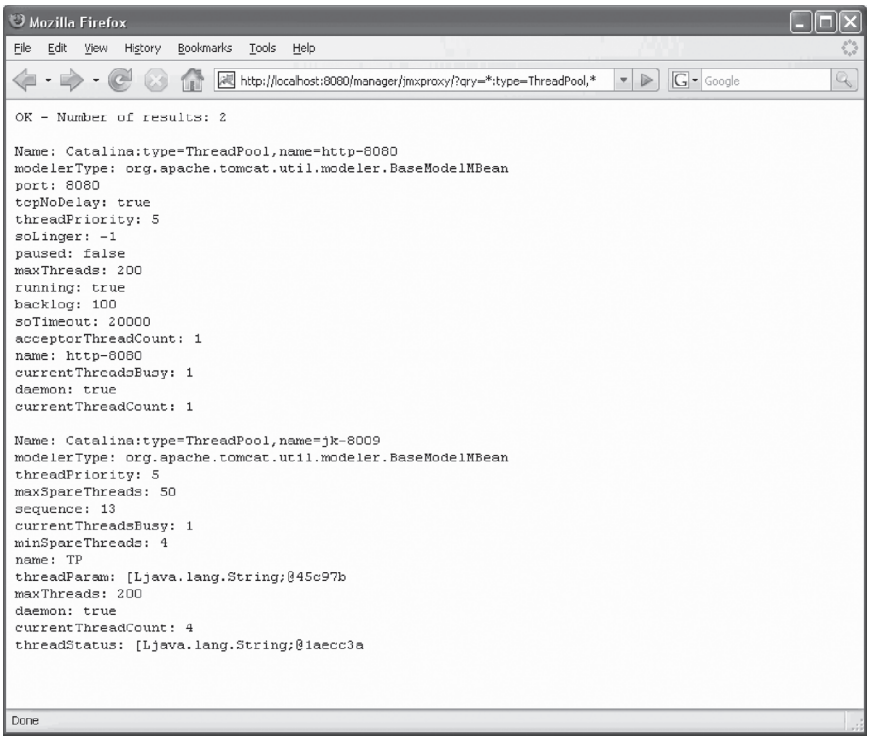

Figure 16-12: Verifying the new value of the maxThreads attribute

The capability to peek into Tomcat internals during runtime and tweak the running server is an extremely attractive feature. Tomcat 6's support for JMX is essential in carrying out these tasks.

The JMX proxy accesses the MBean server within the same JVM. Typical network-management scenarios call for the `management` application to run on an external JVM, and often on another machine over the network. Chapter 8 explains how to perform management tasks in an external JVM running Ant. The next section examines the rich tools support provided by Java SE 6 and how you can take advantage of them in monitoring and managing Tomcat servers.

Using jconsole GUI to Monitor Tomcat

Most administrators create their own script files to perform routine tasks, and the HTML-based access of the Manager application JMX Proxy, as well as the supported Ant tasks, lend themselves well to scripting. For those who would like to monitor and see JMX exposed values in real time, you can use the jconsole utility that is distributed with the JDK since Java 5. The version of jconsole covered in this section is the one distributed with Java SE 6.

You must enable the JMX support inside the Tomcat server. To enable this, in the `<Tomcat Installation>/bin` directory, add a file called `setenv.bat` (or `setenv.sh` in Linux) containing the following line:

```
set CATALINA_OPTS=-Dcom.sun.management.jmxremote
```

You'll need to change the syntax of the environment variable assignment if you are using a Linux shell.

If you don't specify the preceding system property, a Java SE 5 jconsole will not be able to connect to Tomcat at all. In Tomcat 6, jconsole can connect to the JVM but does not see any of the Tomcat-exposed MBeans if the preceding system property is not set.

Connecting to the Local Running Tomcat Instance

With Tomcat 6 running, when you start the jconsole command, you will be prompted to select the JVM to monitor. Look for the local process name of `org.apache.catalina.startup.Bootstrap.start`.

The startup screen for jconsole is shown in Figure 16-13.

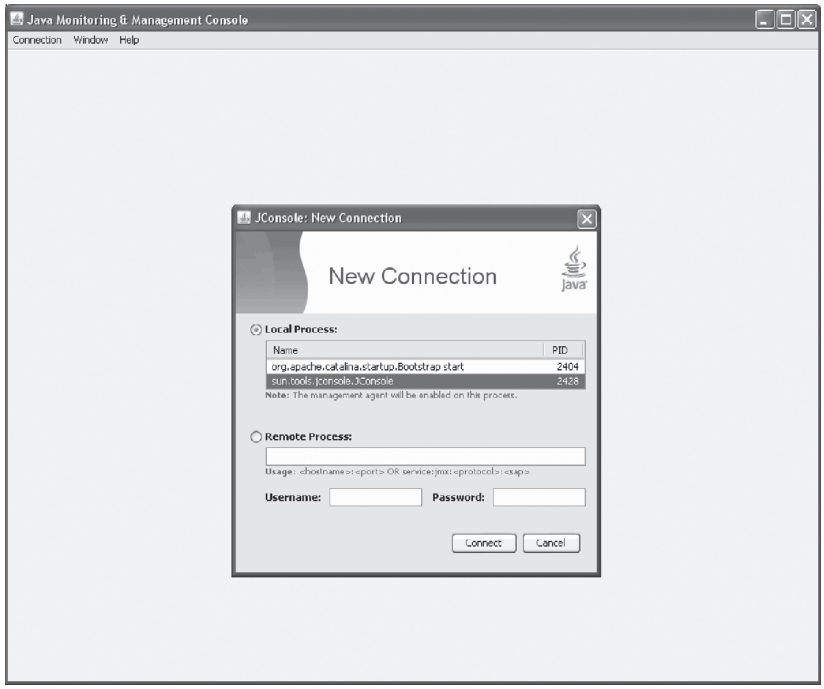

Figure 16-13: Attaching to the Tomcat 6 JVM via jconsole

Click the Connect button after selecting the Tomcat JVM. jconsole connects to the Tomcat JVM and displays all sorts of JMX data exposed by the Java VM itself.

Locating the Tomcat 6 Exposed MBeans

The jconsole displays a lot of exposed metrics courtesy of the fully instrumented JVM. You can find a lot more information about jconsole and the other management and monitoring tools in the JDK by visiting the following URL:

```
http://java.sun.com/javase/6/docs/technotes/guides/management/jconsole.html
```

To locate the MBeans exposed by Tomcat in the jconsole GUI, click the MBeans tab. In this tab, you see JVM MBeans as well as those associated with Tomcat. Those exposed by Tomcat 6 are under the keys named `Catalina` and `Users`.

Figure 16-14 shows jconsole with the Tomcat 6–exposed ThreadPool information. Note the http-8080 and jk-8009 threadpools, the same as the one you saw earlier in the Manager JMXProxy example. jconosle just provides a way to access the same set of Tomcat 6 JMX MBeans.

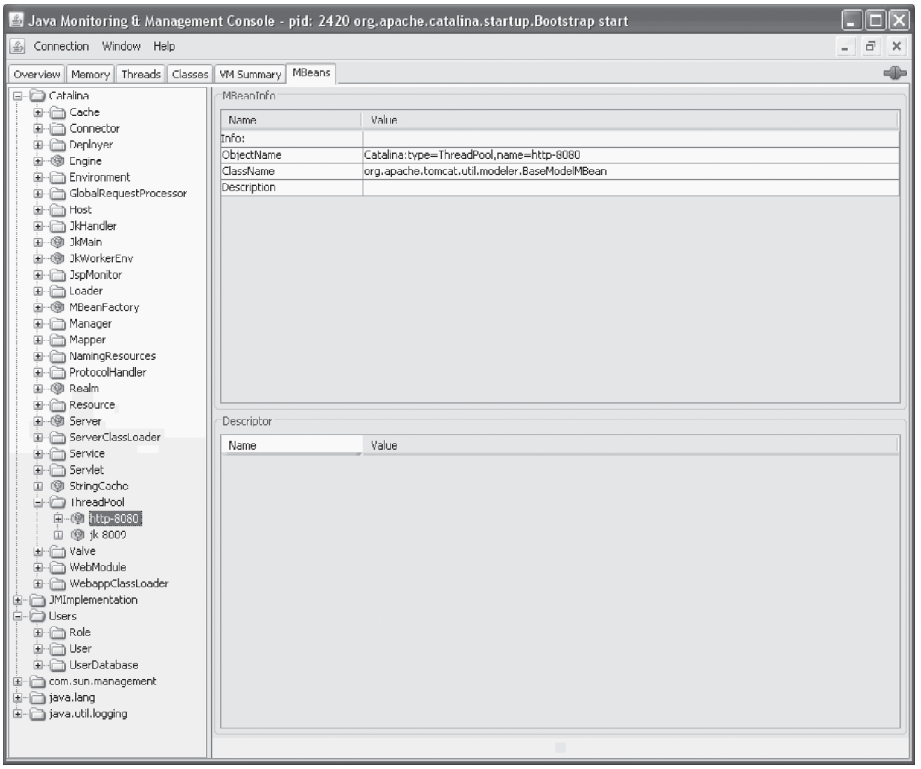

Figure 16-14: Tomcat Exposed MBeans viewed through the jconsole utility

If you examine the list of exposed manageable elements under the `Catalina` key in jconsole, you see that it correspond to the list of JMX manageable elements documented in this chapter. Using jconsole, you can find and monitor the current values of their attributes.

In Figure 16-14, you will notice that users of the Tomcat server are exposed under the `Users` key. All the currently active users on the server, including their passwords and role information, are accessible through this jconsole connection. This is clearly a potential security loop-hole. For this reason, remote management and monitoring are disabled by default. The last section of this chapter shows you how to enable remote Tomcat monitoring and management via jconsole.

Changing Component Attribute Values via jconsole

The writable attributes from the JMX managed component can be changed through the jconsole GUI. As an example, you can change the `maxThreads` property of the http-8080 threadpool from the default of 150 to 200.

In jconsole's MBean tab, look under the Catalina key for the ThreadPool components. Expand the http-8080 component's attributes and then click the `maxThreads` property. You should see the current value of `maxThreads`, as shown in Figure 16-15.

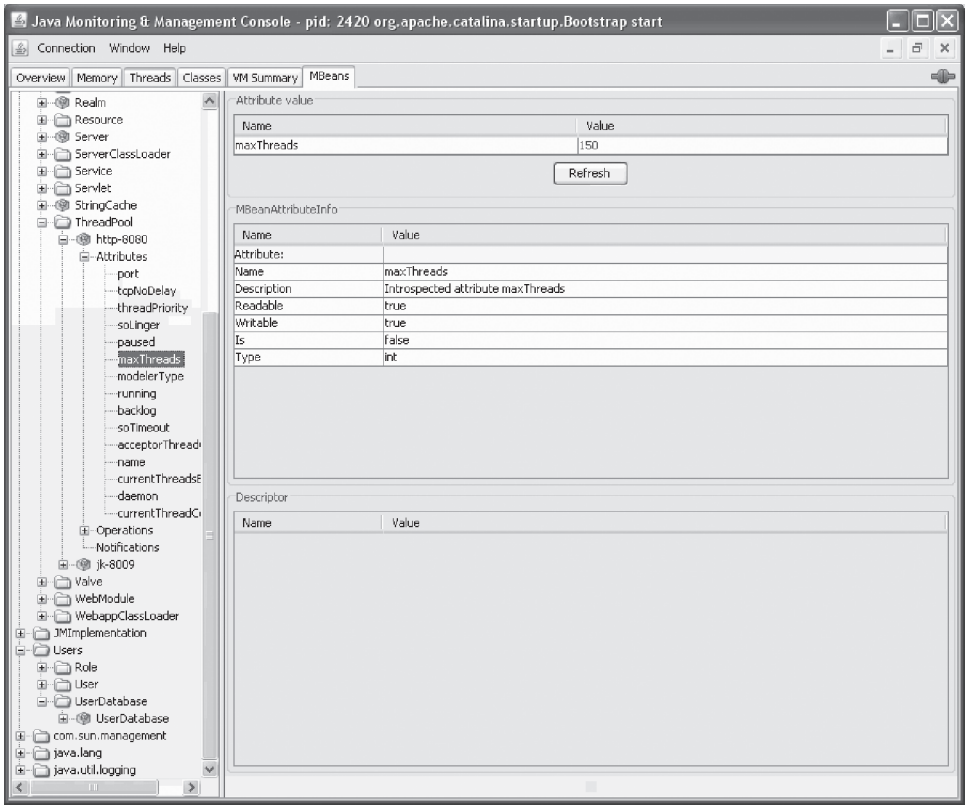

Figure 16-15: Using jconsole to change the value of attributes on a JMX managed component

To change the `maxThreads` value, simply place the cursor on the current value and then change it. Try changing it from the default 150 to 200. After you've changed the value, click the Refresh button to confirm that the change has been made.

Configuring Tomcat for Remote Monitoring

For security reasons, the default Tomcat server configuration has remote JMX–based monitoring disabled. To enable remote monitoring, edit the `setenv.bat` file that you've created in the `<Tomcat installation>\bin` directory to contain the following:

```
set CATALINA_OPTS=-Dcom.sun.management.jmxremote
 -Djava.rmi.server.hostname=192.168.23.2
 -Dcom.sun.management.jmxremote.port=8999
 -Dcom.sun.management.jmxremote.ssl=false
 -Dcom.sun.management.jmxremote.authenticate=false
```

Type all of the preceding code on one single line, and be very careful to avoid typos. You need to replace the `java.rmi.server.hostname` property with your own IP address or host name. On a machine with multiple network connections, you must specify the IP address or host name of the network connection that you are remote managing from. It can be useful to restrict remote management to a specific network connection — for example, the internal intranet only. You can also select an available port for use on your specified connection.

While the previous configuration allows for remote monitoring, it is wide open and insecure. Anyone can connect to the server over the network, if they know the port, using any JMX client. The previous configuration should be enabled only in a test network and never for production.

> *Please be aware that the Manager JMXProxy is by definition remotely accessible because it works through a browser-based interface. In fact, enabling the Manager application in a production environment is a security risk in itself. See Chapter 14 for more information on disabling the Manager application.*

To connect and manage the Tomcat server configured previously, on your client machine start jconsole, select Remote Process, and type **192.168.23.2:8999** (replace with your own host name/ip and management port that you have specified). (See Figure 16-6.)

Once connected, you can view all the Tomcat exposed management components and their attributes and modify the value of writable properties. Jconsole works in exactly the same way remotely as it does with local JVMs.

> *When using Windows, Tomcat must be running on an NTFS formatted volume because of a known bug related to file access permissions.*

Securing the Remote Management Connection with SSL and User Authentication

For maximum security, remote management should not be enabled in production. However, if you must enable remote management, you should at least:

- ❑ Restrict the remote management connection to be within the internal network (or VPN) only
- ❑ Enable authentication
- ❑ Enable SSL transport

You restrict the remote management connection by specifying the `java.rmi.server.hostname` property on the Tomcat instance being managed (shown in the last section).

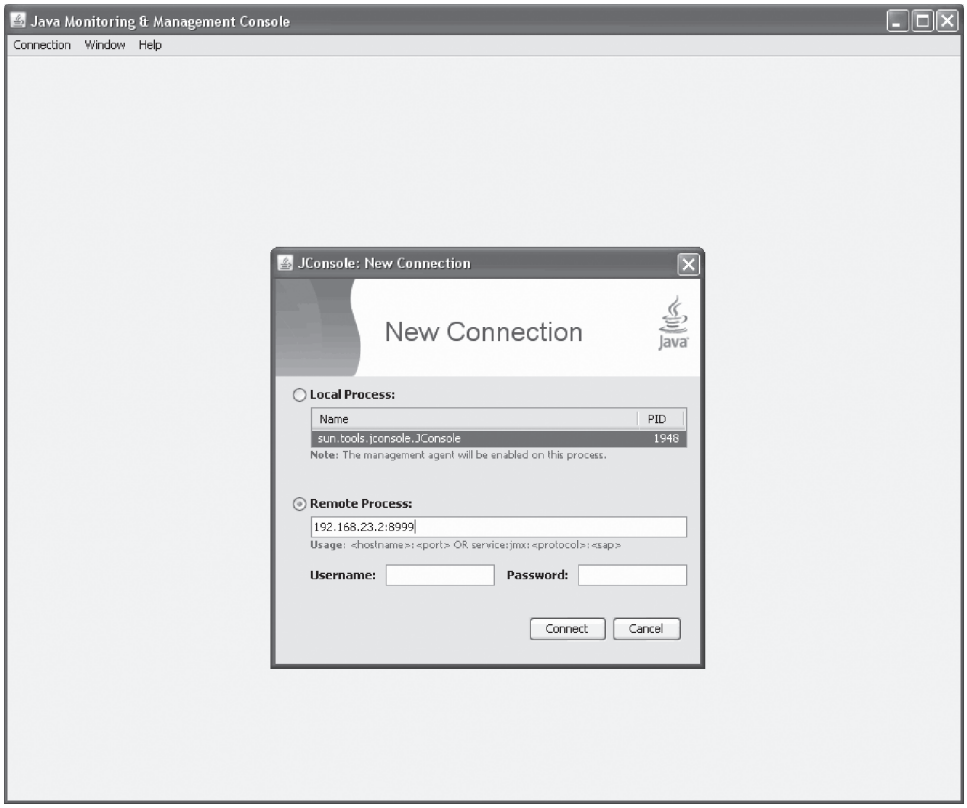

Figure 16-16: Remote monitoring for Tomcat via jconsole

To enable user authentication and SSL transport, modify the `setenv.bat` file to contain:

```
set CATALINA_OPTS=-Dcom.sun.management.jmxremote
 -Djava.rmi.server.hostname=192.168.23.2
 -Dcom.sun.management.jmxremote.port=8999
 -Dcom.sun.management.jmxremote.ssl=true
 -Dcom.sun.management.jmxremote.authenticate=true
 -Dcom.sun.management.jmxremote.password.file=..\conf\password.txt
 -Dcom.sun.management.jmxremote.access.file=..\conf\access.txt
```

You can define roles for monitoring and management in the access control file `conf\access.txt`:

```
monitorRole   readonly
controlRole   readwrite
```

The `monitorRole` can read only the value of attributes exposed by the manageable components. The `controlRole` can read and modify any writable attributes.

For each role, you need to specify the password in the corresponding password file `conf\password.txt`:

```
monitorRole    monpass
controlRole    ctrl1@pass
```

To enable SSL connections for Tomcat, make sure you set `com.sun.management.jmxremote.ssl` to `true`, and then follow the instructions in Chapter 14 to set up SSL transport if you have not already done so.

On the client machine where you intend to run jconsole over SSL, you should have the server's CA cert in your local keystore and start jconsole with:

```
jconsole -Djavax.net.ssl.trustStore=mykeystore
         -Djavax.net.ssl.trustStorePassword=mykeystorepassword
```

You should replace `mykeystore` with the location of your keystore, and `mykeystorepassword` with the password you have set for the keystore.

Summary

To conclude this chapter, let's review some of its key points:

- JMX is a standard for the management of hardware devices, software services, and other manageable entities. The JMX architecture has three levels and each level is componentized. Each level is decoupled from the others.
- The bottom level is instrumentation, and it requires that manageable devices and/or services expose their manageable attributes, operations, and events via a set of MBeans.
- The top level is distributed services. It involves management applications or higher-level agent functionality and is not well defined at this time.
- The middle level is called the agent level. This level aggregates the MBeans from devices and services, and provides a set of services, customized value-added logic, and external/remote access to managed elements.
- Tomcat 6 is fully JMX-compliant. It is also fully instrumented. This means that all of its configuration, internal, and runtime components have MBeans associated with them, enabling these components to be accessed through a JMX agent.
- The Manager application in Tomcat provides a JMX proxy that can be used to access these Tomcat MBeans. The proxy provides a Web interface for querying the MBean Server, and reading and writing MBean attribute values. This interface, provided through an HTTP protocol adapter, can be used to monitor and manage local as well as remote Tomcat instances.
- The jconsole tool included with standard JDK can also be used to monitor and manage Tomcat instances via a user-friendly GUI. jconsole can work with Tomcat instances running on the same machine, or remote instances.
- Remote management should be enabled only on internal test networks. If you must enable remote management in production, do so only after careful consideration for security. Specifically, you should enable user authentication and SSL if you must enable remote management for production systems.
- One major benefit of Tomcat's JMX support is the capability to consolidate the monitoring and management of Tomcat servers via remote JMX access. Ultimately, this centralizes the

management, administration, and monitoring of a large number of Tomcat servers — a requirement that has not been satisfied by Tomcat previously.

In Tomcat 6, one beneficial side effect of componentization and the exposure of internal components via JMX is the capability to configure, start, and operate Tomcat from within an external program or script file (external to the JVM running Tomcat). This is known as the *embedded mode* of operation for Tomcat. Chapter 18 has extensive coverage of this new mode of Tomcat server operation.

Before looking at this new mode of Tomcat server operation, however, let's take a look at Chapter 17, which explores clustering with Tomcat 6.

17

Clustering

The Tomcat server has grown up, from a reference implementation of a Servlet container for demonstrating and testing new APIs to a robust and high-performance Web-tier server. Increasingly, Tomcat is being used in production scenarios to handle real-world Web applications. It is a prime example of a prototype becoming the product. In a sense, the designers and architects realized that their mission changed direction, and they redesigned the Tomcat product for high-stress production deployments.

Tomcat 6 is further along the evolutionary path of improved performance. Real-world deployments place many tough demands on the Tomcat server. Many of these requirements were not important to the Tomcat development team when it was a mere reference implementation. Two such areas are support for *horizontal scalability* (the ability to handle increasing user requests by utilizing a group of physical machines) and *high availability* (the ability to survive hardware or software failures and maintain a high percentage of application uptime). In other words, what happens when there are so many users that Tomcat servers start to crash, and how can you ensure that no user loses data? These are real-world problems, and Tomcat 6 attempts to solve them by providing built-in support for clustering. Clustering, in this context, refers to running multiple instances of the Tomcat server so that it appears to users as a single server.

This chapter covers the many facets of Tomcat 6 clustering, including the following:

- ❏ Basic principles of clustering
- ❏ How Tomcat implements clustering
- ❏ Internal software components that implement Tomcat clustering
- ❏ Technologies that underlie Tomcat clustering
- ❏ Various alternative configurations

Last but not least, the chapter gives you hands-on experience with configuring and experimenting with this exciting feature of Tomcat. You will be working with three different clustered configurations along the way. As an epilogue, the chapter offers some practical suggestions about clustering that may affect your decision to deploy the technology.

Clustering Benefits

Clustering in Tomcat 6 enables a set of Tomcat instances on a LAN (called a *cluster*) to appear to incoming users as a single server. This enables the distribution of work among the servers, called *load balancing*. Chapter 11 covers a load-balancing configuration with the AJP Connector and `mod_proxy` with Apache 2.2.*x* (as well as `mod_jk` for Apache 1.3.*x* and Apache 2.0.*x*). Figure 17-1 illustrates load-balancing.

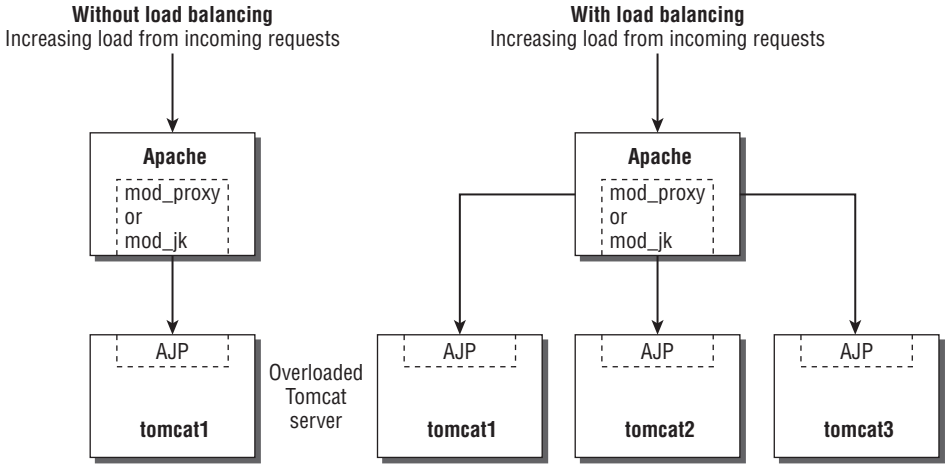

Figure 17-1: Load balancing with Tomcat 6

In Figure 17-1, if an increasing stream of incoming requests were sent to the tomcat1 server, at some point the server will overload and crash by running out of resources. By deploying a load-balanced cluster, however, more requests can be handled.

Scalability and Clustering

Using the load-balanced configuration shown in Figure 17-1, incoming requests are distributed over the tomcat1, tomcat2, and tomcat3 servers. This means that each server is handling only a portion of the incoming requests. If the load-balancing distributor and algorithm are efficient, the system as a whole can handle significantly more requests before overloading.

Scalability refers to the capability to provide service for an increasing number of users. Formerly, scaling an application to more users required an upgrade to expensive, multiple-CPU systems and corresponding memory expansion. This approach is often called *scaling up*. The way that clustering is handled in Tomcat 6 leverages inexpensive high-speed LAN interconnections to share the computing resources of multiple server machines. This approach is called *scaling out* or *horizontal scaling*. It provides an obvious cost advantage, as a server farm of low-cost commodity hardware is less expensive than a single multiple-CPU server.

Load balancing can solve the scalability problem by enabling the cluster to handle significantly more simultaneous requests than a single nonclustered server. Tomcat servers can support horizontal scaling through the use of the AJP Connector and the `mod_jk` plug-in with the Apache Web server.

The Need for High Availability

Another difficult real-world problem that can be solved via horizontally scaled clustering is the *high availability* (HA) issue. The challenge here is avoiding situations in which server software/hardware crashes and becomes unavailable.

In a regular system, all requests being processed by the server are lost, and all users must wait until the server restarts properly before starting their work from the beginning. For example, if the users are online shoppers in a Web store, they will lose the contents of their shopping carts and any data that they may have entered during the checkout process when the server crashed.

This is clearly an unacceptable set of consequences for serious real-world applications. What is desired is a system capable of continuing to handle incoming requests, making a single server crash completely transparent to the end user. The crash and recovery of any individual hardware system should not affect the user experience with the hosted application. Systems with the capability to survive server crashes typically exhibit very high *uptime* or *availability*. These systems are called high-availability (HA) systems.

Tomcat 6 clusters can be used to implement an HA solution. In this scenario, the following occurs:

1. A request that is destined for the crashed server is redirected to another functioning server in the cluster.
2. The original incoming request is processed by the functioning server.
3. The Tomcat server that failed is logically removed from the cluster, so no further requests will be forwarded to it.
4. When and if the crashed Tomcat server recovers, it is logically added back into the cluster, and once again used to handle incoming requests.

The key to enabling this scenario is to realize that any state information maintained by the application in the crashed server (typically, carried in the session) must be somehow made available to the functioning server.

Tomcat 6 provides a workable solution to both the scalability and the HA problem. As such, Tomcat 6 has made major inroads in establishing itself as a serious and robust contender for enterprise-level production deployment.

To better understand how Tomcat 6 achieves scalability and HA, a few words on some basic clustering concepts are appropriate.

Clustering Basics

Several basic system design patterns are found in clustered systems (a specific applied instance of the general-distributed computing problem). This section briefly describes two. The preceding discussion in this chapter and Chapter 11 cover a third: the load-balancing frontend pattern.

Master-Backup Topological Pattern

Figure 17-2 illustrates the master-backup topological pattern. This pattern refers to the way machines are configured and connected to achieve high availability. In this pattern, two (or more) machines are

identically configured in hardware and OS, and they both host the same software. Interconnection exists between the machines (for example, over a LAN). One machine is designated as the *master server* and processes incoming requests. The rest of the machines are *backup servers*. The health of the master server is monitored constantly, either by the backup servers or by an independent hardware/software component. Whenever the master server crashes, one of the backup servers is made the master and request processing continues as if no crash has occurred. This action of the backup server taking over operation after a crash of the master is an instance of the fail-over behavioral pattern, covered in the next section. Master-backup combined with fail-over form the basis of most HA implementations.

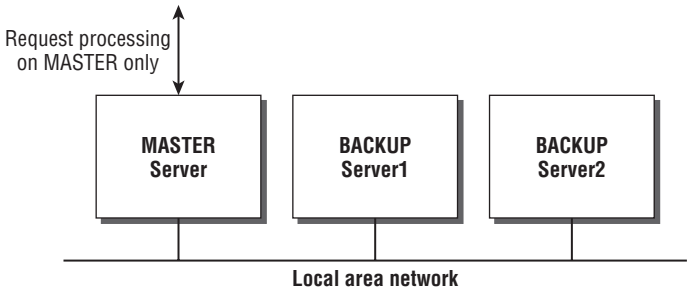

Figure 17-2: The master-backup topological pattern

Fail-Over Behavioral Pattern

Figure 17-3 illustrates the fail-over behavioral pattern. Fail-over occurs when a server crashes in a master-backup system. It refers to the way and means by which the master server is taken over by the former backup server. While in a hardware implementation, this may involve sophisticated switching and communications link isolation; in a software scenario, a state transfer and synchronization mechanism must be in place.

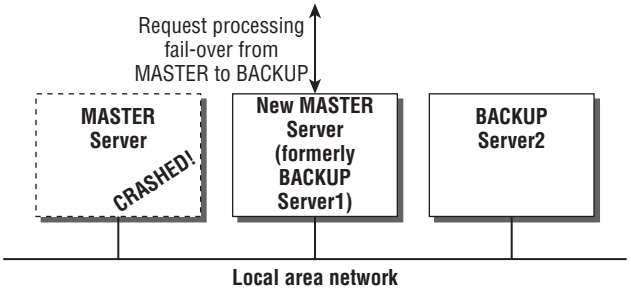

Figure 17-3: The fail-over behavioral pattern

In other words, simply having identically configured hardware, OS, and software applications (as in the master-backup pattern) is not enough to guarantee a logically transparent fail-over.

The backup server taking over the duties of the master server must know "what the master server is up to," and continue from where it left off. This requires the synchronization and sharing of dynamic state information, as shown in Figure 17-3.

Of course, when the master server has crashed, it is too late to query it for information about what it was doing. In all likelihood, the crashed master server is not in any condition to respond.

Therefore, almost all fail-over solutions rely on maintaining and sharing state information between the master and backup servers *before* any crashes occur. Keeping this information current is the only way to ensure that the backup server can take over from where the master server left off during a fail-over.

This state sharing is much tougher to implement than it sounds. State information on a system refers to *any* changes to the system. On a hardware level, this could mean any memory or register write. Imagine having to let all servers in a cluster know about every register and memory write on the master system!

Thankfully, in a Java EE–compliant Servlet/JSP container, a well-accepted convention for tracking state information within Web applications is available. It involves the use of server-side sessions. Tomcat 6's cluster implementation takes advantage of this to provide fail-over capability.

Now it's time to see how Tomcat 6 incorporates the load-balancing, master-backup, and fail-over patterns in its clustering implementation to provide scalable HA features for end users.

Tomcat 6 Clustering Model

This section explores the specific clustering implementation supported by Tomcat 6. Based on the discussion thus far, the implementation can be divided into two layers and various components. Figure 17-4 illustrates this.

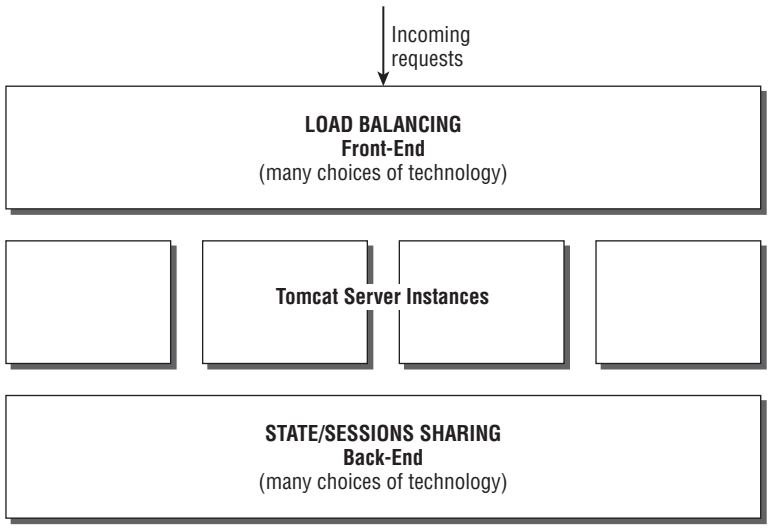

Figure 17-4: Tomcat 6 clustering-implementation architectural model

The two layers that enable clustering are the load-balancing frontend and the state-sharing/synchronization backend. In particular, Tomcat's load-balancing frontend distributes incoming requests to the Tomcat instances, while the backend is concerned with ensuring that shared session data is available to the different instances.

Load Balancing

There are many options for implementing the load-balancing frontend. What you should choose depends on your specific application. These alternatives include (but are not restricted to) the following:

- Round-robin DNS, whereby a domain name resolution results in a set of IP addresses.
- A hardware-based load balancer.
- A software-based load balancer. See *PLB (Pure Load Balancer)*, a popular open-source TCP-based load balancer available from `http://plb.sunsite.dk/`. Another one is `mod_backhand`, from `backhand.org/mod_backhand/`. Yet another software-based load balancer is *balance*, available from `inlab.de/balance.html`.
- Apache `mod_proxy` or `mod_jk` as load balancer.

A discussion and detailed comparison of all of these options is beyond the scope of this book. However, the last option is covered in this chapter because it is the most popular and least expensive option with Tomcat 6 deployments.

mod_proxy/mod_jk Load Balancing and Sticky Sessions

The `mod_proxy` or `mod_jk` load balancer distributes incoming requests in a round-robin manner among the available Tomcat workers, but will also respect a *load factor* that you can specify. In addition, these load balancers support sticky sessions.

Understanding Sticky Sessions

When sticky sessions (or *session affinity*) is enabled on `mod_proxy` or `mod_jk`, it ensures that all incoming requests with the same session are routed to the same Tomcat 6 worker. Figure 17-5 illustrates this concept.

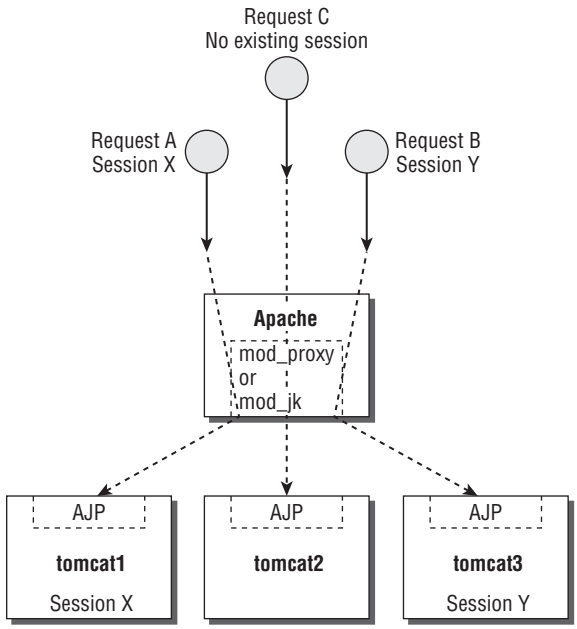

Figure 17-5: mod_jk2 load balancing with session affinity

In Figure 17-5, incoming request A is routed to tomcat1 because session X is created and maintained on the tomcat1 instance. Meanwhile, request B is routed to tomcat3 because session Y is created and maintained on the tomcat3 instance. Request C has no session, so it is routed to the next server in the round-robin distribution, which is tomcat2.

This is a highly functional clustering configuration that is relatively easy to configure, and it can be used to scale a Web application across a cluster of Tomcat 6 servers.

The only disadvantage of this configuration is that sessions are Tomcat instance–specific. If a Tomcat instance is lost, all its sessions are lost.

For example, in Figure 17-5, if the tomcat1 instance crashes, session X is lost forever. Request A will not be handled by the system, and the user will lose the session (and whatever was being done at the time the failure occurred). However, if there is some way to help the clustered Tomcat instances to share session information, then one of the other Tomcat instances still up and running can take over and service incoming requests with sessions created by the crashed server.

Session Sharing

As with the load-balancing frontend, you have numerous session-sharing backends from which to choose. Each provides a different level of functionality, as well as implementation complexity.

Session sharing is the secret behind most implementations of an application server fail-over mechanism. It ensures transparent transfer of the sessions that were being handled by the crashed server.

In the following discussion, we assume that the `mod_proxy` or `mod_jk` load balancing frontend is used. These are the most popular production configurations for clustered Tomcat 6 instances. The available session-sharing configuration options include the following:

- Sticky sessions with no session sharing
- Sticky sessions with a Persistent Session Manager and a shared file store
- Sticky sessions with a Persistent Session Manager and a JDBC store to RDBMS
- In-memory session replication

Sticky Sessions with No Clustered Session Sharing

This is the scenario that was tested in Chapter 11, when `mod_jk` and an AJP Connector were used to round-robin requests amongst a cluster of Tomcat 6 server instances. In this scenario, the `mod_jk lb` (load balance) worker ensures that requests destined for the same session are always handled by the same Tomcat worker instance. The session ID is encoded with the route name of the server instance that created it, assisting in the routing of the request.

While this setup may sound contrived, it is extremely practical and pragmatic in many production scenarios. The advantages of this simple clustering setup include the following:

- Application scalability through round-robin load balancing (new sessions are always created on the next worker in the round-robin queue)
- Simplicity in setup and maintenance (the `mod_jk lb` worker will detect crashed servers and reinstate recovered ones)
- No additional configuration or resource overhead (as no session sharing is occurring)

The major disadvantage is the lack of HA features. A crashed server means lost sessions.

In situations in which server crashes are rare occurrences, and when session losses during these rare occasions are acceptable, this should be the clustering solution deployed.

Sticky Sessions with a Persistence Manager and a Shared File Store

Tomcat 6 is packaged with a Persistence Manager component that can be configured into any application context. Chapter 6 covers the configuration of this component. The main purpose of persistence management is to provide continuity to sessions when a server shuts down and is restarted. Because the sessions are persisted to either the disk or an RDBMS (via a File Store or a JDBC Store component), they can have a lifecycle that is longer than the server's. In addition, because sessions can be configured to be "swapped out to the store" after a specified idle time, the Persistence Manager also provides a form of protection against system crashes. (That is, any persistent session can be recovered when the system restarts after a crash.)

Note that the Persistence Manager in Tomcat 6 is designed with no consideration for clustering. It deals only with the lifecycle and session of one single Tomcat instance. However, by sharing the store between multiple Tomcat 6 server instances (via either a shared file system or an RDBMS), a certain level of session sharing can be accomplished.

Figure 17-6 illustrates the Persistence Manager with a shared file store. In this case, all the Tomcat server instances use the same directory to store their sessions. This directory is accessible from all the servers via the OS's shared file mechanism (NFS, SMB, and so on). Now, any sessions created by any Tomcat instance will *eventually* be visible to the other instances. Any modifications made by any instance will also be *eventually* visible to the other instances. It is important to reiterate the importance of the word *eventually* here because the Persistent Session Manager does not guarantee exactly when a session will be persisted to the store (either after creation or modification). Until a session (or a change in the session) is persisted to the store, that information is not available to the other instances.

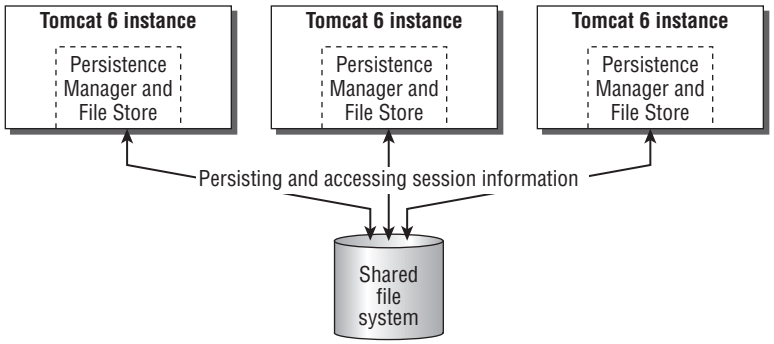
Tomcat 6 instance
Persistence Manager and File Store
Tomcat 6 instance
Persistence Manager and File Store
Tomcat 6 instance
Persistence Manager and File Store
Persisting and accessing session information
Shared file system

Figure 17-6: Tomcat 6's Persistence Manager with a shared file store

At this point, you may be wondering what good a clustering system is that *eventually* shares its session information.

There is actually a very good and pragmatic answer. It is *slightly better* than the sticky session with *no* session sharing solution previously discussed.

In clustering situations where server crashes happen very infrequently, using this scheme can further minimize the loss of sessions during the moment of system crash and fail-over. This is because any

persisted session at fail-over time can be immediately handled by any one of the remaining servers in the cluster.

Note that sticky sessions must still be configured in the `mod_jk` with this scheme. This means that most of the time, a session will be serviced throughout its lifetime by the same Tomcat instance. The only exception occurs when the original server crashed during the session's lifetime. Sticky sessions also increase the probability that the session will be persisted to the store. This is because the longer the session lingers around, the greater the probability of it being persisted.

The advantages of this session-sharing scheme include the following:

- Application scalability through round-robin load balancing (new sessions are always created on the next worker in the round-robin queue).
- Relatively easy setup and maintenance. The `mod_jk lb` (load balance) worker will detect crashed servers and reinstate recovered ones.
- It provides a measure of HA in most situations because any persisted session is shared and can be handled by another server in the cluster.

The disadvantages of this session-sharing scheme include the following:

- Some sessions may still be lost during fail-over.
- Access traffic on the network supporting the shared file system can be heavy in a highly loaded server cluster.

Sticky Sessions with a Persistent Session Manager and a JDBC-Based Store

Note in Figure 17-6 that there is no conceptual difference between this session-sharing scheme and the previous one. In fact, they both use the same Persistent Session Manager component. In this case, JDBC is used to persist session information onto an RDBMS. The set of benefits and weaknesses remains identical to the previous scheme.

The only additional benefit of going to a JDBC-based scheme is the potential performance improvement on systems persisting a large number of sessions. If the applications running on the cluster are using the same RDBMS as the Persistent Session Manager, however, this slight performance edge may disappear (because of increased contention).

In-Memory Session Replication

Unlike the other two session-sharing schemes, this session-sharing mechanism is not built on top of a shared persistent storage. With in-memory session replication, session information is maintained in synchronization within the memory, across the clustered server instances.

Two replication patterns are supported by Tomcat 6. In the first replication pattern, all sessions are replicated between all server instances. In the second pattern, sessions are replicated between only a server and its backup instance, regardless of how many servers are on the connected network cluster.

Because two or more instances share the same session information, these session replication mechanisms have the potential to provide the full benefits associated with clustering. Figure 17-7 illustrates how this is accomplished.

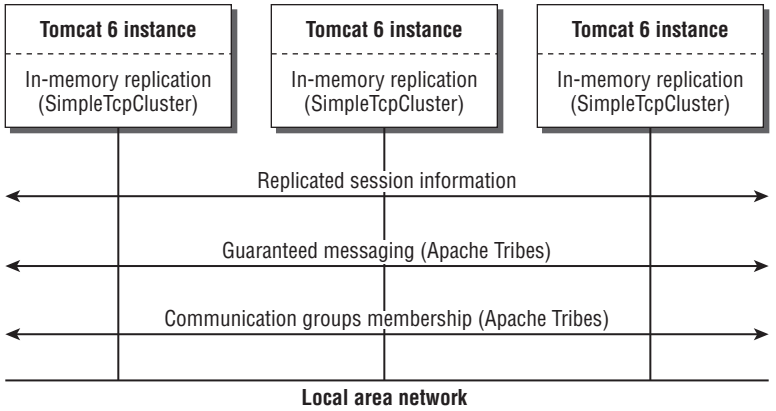

Figure 17-7: Tomcat 6 in-memory session replication

In Figure 17-7, the Tomcat 6 server instances running in the cluster are implemented as a communications group. At the Tomcat instance level, the cluster implementation is an instance of the `SimpleTcpCluster` class.

Depending on your needs, and the replication pattern you want, you can configure `SimpleTcpCluster` with one of two managers. The two available managers with Tomcat 6 are:

- `DeltaManager` to replicate sessions across all Tomcat instances
- `BackupManager` to replicate sessions from a master to a backup instance (see the master-backup topological discussion earlier in this chapter)

Under the hood, `SimpleTcpCluster` uses Apache Tribes to maintain and communicate with the communications group. Apache Tribes is a reusable API library for implementation of communications groups over TCP networks.

Group membership is established and maintained by Apache Tribes. If a server crashes, it is automatically removed from the group. A recovered server is automatically made a member of the group again.

Apache Tribes also offer several levels of guaranteed message delivery between group members. This capability is utilized by the `SimpleTcpCluster` to provide a configurable level of service.

Any session-creation and modification activities on Tomcat 6 instances are sent to one or all instances within the communications group (depending on the manager implementation configured). The receiving group member(s) then update its (their) own session image in memory to reflect the change. In this way, sessions and changes are replicated immediately — in memory, and between members of the group.

The main benefits of this approach are as follows:

- Full round-robin load balancing enables an even distribution of requests (subject to prefigured `lbfactor` load factor on `mod_jk`). Any server can handle any request.
- With full HA support, any server can crash and fail-over with no session loss.

The second item is the major benefit delivered by the in-memory session replication scheme. The other schemes examined thus far cannot deliver this benefit.

However, with the benefit comes substantial cost. Here are the items of concern for this approach:

- The traffic on the interconnection (usually a LAN) can quickly become very heavy, especially when you are using `DeltaManager`, where changes to all sessions are sent to all members of the group.
- Sessions are *not* persistent. This means that this model assumes that the overall system operates continuously (sometimes called *24/7* or *nonstop operation*). All sessions are lost if the entire cluster is shut down. This is not the case with persistent session management.
- Configuration, tuning, and maintenance can be quite complex.

The remaining sections in this chapter cover each of the clustering mechanisms in more depth, and provide hands-on configuration with working examples.

Working with Tomcat 6 Clustering

As discussed earlier, a Tomcat 6 clustering implementation depends on a load-balancing frontend and a session-sharing backend. The load-balancing frontend may implement sticky sessions (using Apache `mod_proxy` or `mod_jk`), which ensure that the same clustered Tomcat 6 instance will always handle the same session. Taking a peek under the hood reveals why this is very important in several clustering configurations.

Session Management in Tomcat 6

Sessions are created and managed by the Tomcat 6 container during application execution, and are made available to JSP and servlets via the application context. In a single-server instance, Tomcat 6 sessions are objects (which can contain and reference other objects) that are kept on behalf of a client.

Because the HTTP protocol is stateless, there is no simple way to maintain application state using the protocol alone. For example, consider a shopping cart application. Each product page accessed by a user comes into the server as a separate and distinct HTTP request. There is no way for the server to match up independent incoming requests that represent an application flow.

However, most Web applications need to maintain state information associated with a user (for example, the items in the user's shopping cart). A server-side session is the main mechanism used to maintain state. It works as follows:

1. The server writes a cookie to the user's browser instance. The cookie contains a token to retrieve the server-side session (data structure).
2. The cookie is supplied by the browser instance every time it accesses a page on the site.
3. The server reads the token in the cookie to extract the corresponding session.

An analogy to a session is the coat check tag that one may obtain prior to entering a theatre or concert. In this case, the cookie is the tag (smaller and simpler to carry) and the session is the coat (larger, but important data that is kept on the server side). You, the client, hold the tag and return the tag for the

coat, which the establishment holds on your behalf. The browser client holds the cookie, and returns the cookie each time in a connection to the server. Using the cookie, the server is able to locate the session on which the browser client is working.

For browsers that do not support cookies, it is possible to use *URL rewrite* to achieve a similar effect. In URL rewrite, any URL that is being supplied by the application is decorated with the session ID being used. This enables the Web application to extract the session ID from the incoming URL during runtime.

The Role of Cookies and Modern Browsers

All popular modern-day browsers (Firefox, Internet Explorer, Opera, Mozilla, and so on) support the use of cookies. Cookies are managed on the browser's host PC, and indexed by the Web site's host name. In addition, all modern-day browsers support multiple concurrent connections to the same server. For example, you can start as many instances of Internet Explorer as you want (subject to machine resource constraints) and have them all connect to the `www.wrox.com` URL.

```
www.wrox.com/
```

Each instance you start manages its own client-side session. This is not to be confused with server-side sessions. In essence, when you start multiple instances of a browser pointing to the same server, it appears to the server as if different users are accessing it (each instance manages its own copy of a cookie from the server). In other words, each client-side browser instance will have its own independent, associated server-side session.

> *Note that if a load-balancing mechanism redirects an incoming request to a different host, the cookie supplied will be different (because cookies are indexed by host names) and the session information will not be maintained.*

Configuring a Tomcat 6 Cluster

This section describes the configuration of an actual Tomcat 6 cluster. The cluster consists of three independent Tomcat 6 instances, and makes use of the following:

- `mod_jk` load-balancing frontend
- In-memory session replication backend

This configuration is similar to the one featured in the AJP Connector load-balancing example presented in Chapter 11. The main difference is in the use of multiple `%CATALINA_BASE%` settings (`$CATALINA_BASE` on Unix/Linux) for each Tomcat instance, and the cluster naming of the server instances.

Ideally, the following configuration experiments should be performed on an actual cluster of physical machines running Tomcat 6 on a network. However, not everyone has access to such extensive hardware. To provide all readers with a hands-on configuration experience, the example utilizes multiple instances of Tomcat 6 running on the same machine.

> *If you do configure multiple machines, make sure that their clocks are synchronized. This can be done by synchronizing the machines' clocks with an NTP/Internet time server. Read the manual of your operating system for the specifics.*

Setting Up Multiple Tomcat Instances on One Machine

To enable multiple instances of Tomcat 6 to run on the same physical machine, each instance must have at least the following:

- Its own configuration directory
- Its own `temp` directory
- Its own `webapps` directory
- Its own temporary `work` directory
- Its own `logs` directory
- TCP ports (for the AJP Connector) that do not conflict with other instances
- Optionally, other private TCP or JDBC resources, depending on the backend session-sharing mechanism being deployed

Three batch files, called `start1.bat`, `start2.bat`, and `start3.bat`, are created and placed into the `<Tomcat Installation Directory>/bin` directory.

These batch files set the `CATALINA_BASE` environment variable and then call the `startup.bat` Tomcat startup script. Tomcat 6 checks for the existence of the `CATALINA_BASE` environment variable and uses it to locate the base directory for startup. Each of the `start1.bat`, `start2.bat`, and `start3.bat` files sets the `CATALINA_BASE` variable to a different directory, allowing for variation in configuration. For example, `start1.bat` contains the following:

```
set CATALINA_BASE=c:\cluster\machine1
call startup
```

On Linux systems with a Bourne shell, it contains the following:

```
export CATALINA_BASE=/cluster/machine1
sh startup.sh
```

This tells Tomcat 6 to look for configuration information and Web application in the `c:\cluster\machine1` directory. Figure 17-8 shows the directory hierarchy used here in subsequent cluster testing.

Note in Figure 17-8 that each machine1, machine2, and machine3 directory houses the configuration files for the respective Tomcat 6 instances to be started.

Shutting Down the Tomcat Cluster

Similar to the `startn.bat` files, three `stopn.bat` files exist for shutting down the individual server instances. These batch files set the `CATALINA_BASE` environment variable and then call the shutdown script for the server. For example, following is the content of `stop3.bat`:

```
set CATALINA_BASE=c:\cluster\machine3
call shutdown
```

Configuring Minimal Web Applications

Only the `examples` application (the servlet and JSP examples from Tomcat distribution) is loaded for each of the three machine instances. In general, when setting up a cluster, you should try to minimize the

number of applications loaded. This is because considerable overhead is associated with each clustered application (session management, network traffic, and so on). The `webapps` subdirectory on clustered machines should be thoroughly clean, except for the clustered Web applications.

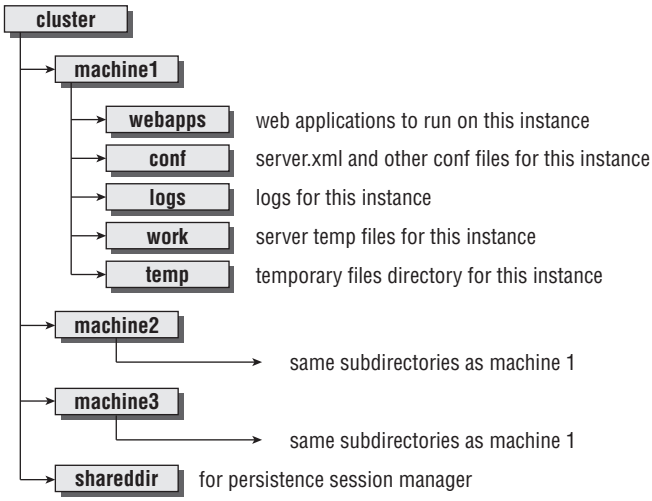

Figure 17-8: Directory tree for the clustering examples

In this case, make sure you copy the `webapps\examples` subdirectory from the Tomcat 6 distribution into each of the `webapps` directories of the three machines.

Disabling the HTTP Connectors

The default `server.xml` file included with the Tomcat 6 distribution sets up two Connectors. One is an HTTP 1.1 Connector listening on port 8080, and the other one is an AJP 1.3 Connector listening on port 8009. You must first comment out the HTTP Connector if you are using the standard `server.xml` file. This ensures that the three instances will not fight for the 8080 port during startup:

```
<Server port="8005" shutdown="SHUTDOWN>
...
<Service name="Catalina">
 <!--
    <Connector port="8080" protocol="HTTP/1.1"
               maxThreads="150" connectionTimeout="20000"
               redirectPort="8443" />
  -->
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
...
```

Note that if you use the code distribution provided with this book, the HTTP Connectors are already removed from the `server.xml` files.

Configuring AJP TCP Ports for Clustered Tomcat Instances

In order for the three Tomcat 6 instances to coexist peacefully on a single physical machine, it is necessary to give the AJP Connectors different TCP ports on which to listen. By default, the server is listening

on port 8005 for shutdown, and the AJP Coyote Connector listens on 8009. The relevant lines in `server.xml` that must be customized for each instance are highlighted here:

```
<Server port="8005" shutdown="SHUTDOWN">
  <Listener className=
            "org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
  <Listener className="org.apache.catalina.core.JasperListener" />
  <Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" />
  <Listener className=
            "org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
  <GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"
              type="org.apache.catalina.UserDatabase"
              description="User database that can be updated and saved"
              factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
              pathname="conf/tomcat-users.xml" />
  </GlobalNamingResources>
  ...
  <Service name="Catalina">
  <Connector  port="8009" protocol="AJP/1.3"/>
   ...
```

The following table shows what you need to configure in each instance.

Instance Name	File to Modify	TCP Ports (Shutdown, AJP Connector)
machine1	\cluster\machine1\conf\server.xml	8005, 8009
machine2	\cluster\machine2\conf\server.xml	8105, 8109
machine3	\cluster\machine3\conf\server.xml	8205, 8209

The settings chosen here ensure that there will be no conflict starting the three Tomcat 6 instances simultaneously on the same physical machine. If you are actually setting up the test across three physical machines on the network, they can all use the setting of `machine1` in the table.

Setting Up jvmRoute for Each Tomcat Instance to Support mod_jk

To support the `mod_jk` load balancing, you must set a unique `jvmRoute` for each Tomcat instance. `jvmRoute` is an Engine attribute and acts as an identifier for that particular Tomcat worker. This identifier must be unique across all the available Tomcat instances participating in the load-balancing environment. It must also be specified in the Apache Server's `mod_jk`'s `worker.properties` file (see next section).

In each of the `server.xml` files, modify the following highlighted line to add the `jvmRoute` attribute.

```
...
<Service name="Catalina">
<Connector  port="8009" protocol="AJP/1.3"/>
...
```

(continued)

```
      <Engine name="Catalina" defaultHost="localhost" jvmRoute="machine1">
    <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
              resourceName="UserDatabase"/>
    ...
```

The following table shows the `jvmRoute` attribute values that you need to set for each of the three Tomcat instances.

Instance Name	File to Modify	jvmRoute Value
machine1	\cluster\machine1\conf\server.xml	machine1
machine2	\cluster\machine2\conf\server.xml	machine2
machine3	\cluster\machine3\conf\server.xml	machine3

If you are configuring on three physical machines, you must configure the Tomcat instance on each machine with a different `jvmRoute` value.

Setting the <distributable> Attribute for Web Applications

Instead of creating a Web application from scratch, this experiment takes advantage of the existing `jsp-examples` Web application. To indicate to the Servlet container (Tomcat 6) that this application can be clustered, a Servlet 2.4 standard `<distributable>` element is placed into the application's deployment descriptor: the `web.xml` file. The following code shows the placement of the `<distributable>` element. If you remove this element, the session maintained by this application across the three Tomcat 6 instances will not be shared:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
   version="2.5">
   <description>
      Servlet and JSP Examples.
   </description>
   <display-name>Servlet and JSP Examples</display-name>
   <distributable/>
    <!-- Define servlet-mapped and path-mapped example filters -->
    <filter>
        <filter-name>Servlet Mapped Filter</filter-name>
        <filter-class>filters.ExampleFilter</filter-class>
    ...
```

This element must be manually added to the `web.xml` file in all three instances:

- `\cluster\machine1\webapps\examples\WEB-INF\web.xml`
- `\cluster\machine2\webapps\examples\WEB-INF\web.xml`
- `\cluster\machine3\webapps\examples\WEB-INF\web.xml`

If you are configuring three physical machines, make sure that the `web.xml` file on each machine has the `<distributable/>` element added.

Configuration Consistency

The three clustered instances of Tomcat 6 should be identically configured, both software and hardware. This is a wise practice, in general, to reduce potential problems arising from dissimilar configuration. Typically, if the application or system requires machines with different hardware/software configurations, they are maintained in separate clusters (or in a nonclustered configuration).

Common Front End: Load Balancing via Apache mod_jk

The load-balancing front end consists of an Apache server with `mod_jk` installed. The following is only a brief recap of the configuration procedure. See Chapter 11 for a detailed, step-by-step explanation of this configuration.

Apache Server Configuration

Make sure you are using the version of the `mod_jk.so` that supports your Apache server. Many problems may arise from version mismatch. The examples in this chapter are tested with Apache 2.2.3 (Win32) with `mod_jk 1.2.19`.

On the Apache server side, if the `mod_jk` support is not included, you must make sure the `mod_jk` module is loaded at startup. This can be done by adding the following line to the `conf/http.conf` file:

```
LoadModule jk_module modules/mod_jk.so
```

This line should immediately follow all the `LoadModule` directives in the file. Note that the downloaded `mod_jk` library needs to be renamed to `mod_jk.so`. It should be placed in the `modules` directory of the Apache server.

mod_jk Configuration

When the `mod_jk` module starts up, it looks for a `workers.properties` file. The following `workers.properties` file should be placed into the `conf` subdirectory of the Apache server:

```
worker.list = bal1,stat1
worker.machine1.type = ajp13
worker.machine1.host = 192.168.23.2
worker.machine1.port = 8009
worker.machine1.lbfactor = 10
worker.machine2.type = ajp13
worker.machine2.host = 192.168.23.2
worker.machine2.port = 8109
worker.machine2.lbfactor = 10
worker.machine3.type = ajp13
worker.machine3.host = 192.168.23.2
worker.machine3.port = 8209
worker.machine3.lbfactor = 10
worker.bal1.type = lb
```

(continued)

```
worker.bal1.sticky_session = 1
worker.bal1.balance_workers = machine1, machine2, machine3
worker.stat1.type = status
```

You need to replace `192.168.23.2` with the IP address of your machine (or machines if you are using more than one). For a detailed explanation of the configuration directives, refer to Chapter 11.

Map Path to Load Balance Working Using JkMount

Back in the `http.conf` file of the Apache server, you need to map the path `/examples/jsp/*` to the `mod_jk` load balance worker that you have defined in the `workers.properties` file.

```
JkMount /examples/jsp/*  bal1
JkMount /jkstatus/       stat1
```

You also need to make sure an entry exists to tell the Apache server where the `workers.properties` file is located.

```
JkWorkersFile   conf/workers.properties
```

Preparation for Using Different Session-Sharing Backends

This completes the basic common setup for the upcoming cluster examples. Each of the session-sharing backends that will be configured requires very specific configuration and customization, and each is covered individually. More specifically, the following example shows how to configure the following backends:

- In-memory replication
- The Persistent Session Manager, using a shared-file system
- The Persistent Session Manager, using JDBC-to-MySQL RDBMSs

The following section covers the configuration of the in-memory replication mechanism.

Backend 1: In-Memory Replication Configuration

Two components need to be configured to enable in-memory configuration with Tomcat 6. Figure 17-9 depicts the position and function of the two components.

In Figure 17-9, a new `<Cluster>` component is responsible for the actual session replication. This includes the sending of new session information to the group, incorporating new incoming session information locally, and management of group membership. Under the hood, it uses the Apache Tribe group communications framework to get the job done. Another component, a replication Valve, is used to reduce the potential session replication traffic by ruling out (filtering) certain requests from session replication.

Operation of the Tomcat 6 SimpleTcpCluster

The default implementation of in-memory replication for Tomcat 6 is called `SimpleTcpCluster`. It is the only one available as of this writing. Development for clustering and the underlying Apache Tribes groups communications framework is an ongoing development activity.

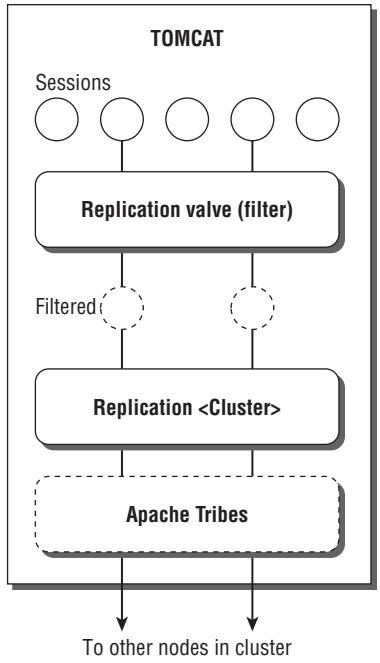

Figure 17-9: Tomcat components for in-memory session replication

This operation of Apache Tribes–based `<Cluster>` implementation is shown in Figure 17-10. It uses regular multicast (heartbeat packets) to determine membership (and TCP for data transfer and other communications). Any node that is up and running must multicast a heartbeat at regular *frequency*. Nodes within the same cluster listen to and multicast at the same multicast address and port. Nodes that do not multicast within a required *dropTime* are considered dead and are removed from the cluster (until they start multicasting again). The membership listens for the multicast and manages the set of current nodes in the cluster dynamically.

Session replication requests and session updates are sent between member nodes in the cluster using TCP connections directly, in an end-to-end connection. This means that a node sending replication data makes a direct TCP connection to each and every member node in the cluster when replicating a session.

Because of the amount of network traffic generated, the default implementation is practical for use only for clusters with a small membership size. You can achieve some economy by using the `BackupManager` instead of the default `DeltaManager` with the `<Cluster>` component. The `BackupManager` sends data to a backup node (and nodes with the corresponding application deployed) instead of to all members in the cluster.

Cluster Session Manager Configuration with the <Cluster> Element

The first component is a new `<Cluster>` component. For this example, the component is nested inside an enclosing `<Host>` element. Including this component in a `<Host>` essentially enables session replication for all applications in the host. The standard manager component used to manage sessions in Tomcat will be replaced with a session replication–enabled manager (either `DeltaManager` or `BackupManager`).

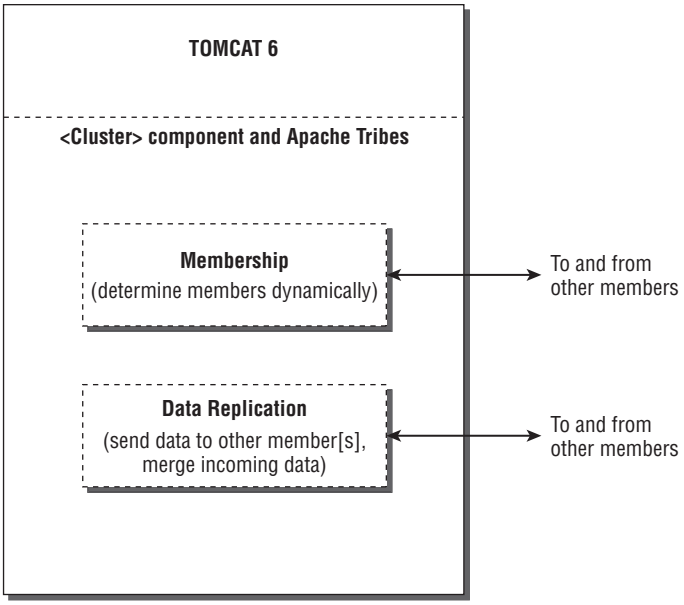

Figure 17-10: Operational model of Tomcat 6 SimpleTcpCluster implementation

Alternatively, if you were to nest the `<Cluster>` element under the `<Engine>` element, all deployed applications in all hosts are session replicated. Here is the configuration for the `<Cluster>` element:

```
<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
             channelSendOptions="8">
      <Manager className="org.apache.catalina.ha.session.DeltaManager"
               expireSessionsOnShutdown="false"
               notifyListenersOnReplication="true"/>
      <Channel className="org.apache.catalina.tribes.group.GroupChannel">
        <Membership className="org.apache.catalina.tribes.membership.McastService"
                    address="228.0.0.8"
bind="192.168.23.2"
                    port="45564"
                    frequency="500"
                    dropTime="3000"/>
        <Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"
                  address="192.168.23.2"
                  port="4200"
                  autoBind="100"
                  selectorTimeout="5000"
                  maxThreads="6"/>
   <Sender className="org.apache.catalina.tribes.transport.ReplicationTransmitter">
   <Transport className=
"org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>
        </Sender>
        <Interceptor className=
"org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>
```

```
        <Interceptor className=
"org.apache.catalina.tribes.group.interceptors.MessageDispatch15Interceptor"/>
      </Channel>
    <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"
 filter=".*\.gif;.*\.js;.*\.jpg;.*\.htm;.*\.html;.*\.txt;"/>
    <Valve className="org.apache.catalina.ha.session.JvmRouteBinderValve"/>
       <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
                 tempDir="/tmp/war-temp/"
                 deployDir="/tmp/war-deploy/"
                 watchDir="/tmp/war-listen/"
                 watchEnabled="false"/>
       <ClusterListener className=
"org.apache.catalina.ha.session.JvmRouteSessionIDBinderListener"/>
       <ClusterListener className=
"org.apache.catalina.ha.session.ClusterSessionListener"/>
        </Cluster>
```

Replace `192.168.23.2` with your own IP address. Note that the highlighted line is the TCP port on which the Apache Tribes data replication mechanism listens for incoming session update information. Because all three of the Tomcat instances are running on the same physical machines, the port must be different for machine1, machine2, and machine3. The following table shows the port assignments that you need to use. Make sure you configure them properly in the corresponding `server.xml` file.

Instance Name	tcpListenPort
`machine1`	4200
`machine2`	4300
`machine3`	4400

Binding to a Specific Adapter on Multi-Homed Machines

Another parameter to be careful about is the `bind` attribute of the `<Membership>` element. This `bind` attribute is highlighted in bold within the preceding `<Cluster>` configuration listing. This attribute selects the physical network interface to use on machines that have more than one network adapter — called multi-homed machines. If you have only one network adapter, this parameter is typically not necessary. However, because of the way operating systems tend to create virtual network adapters for high-speed links such as IEEE 1394 Firewire, specifying this parameter prevents surprises. The address specified should be the IP address of the specific adapter you want to use.

Configuring the <Cluster> Element

The following table shows the attributes available for the `<Cluster>` element.

Attribute Name	Description	Default
`className`	The implementation Java class for the cluster manager. Currently, only `org.apache.catalina.ha.tcp.SimpleTcpCluster` is available.	

Table continued on following page

Attribute Name	Description	Default
channelSendOptions	Option flags are included with messages sent and can be used to trigger Apache Tribes channel interceptors. The numerical value is a logical OR of flag values including: Channel.SEND_OPTIONS_ASYNCHRONUS 8 Channel.SEND_OPTIONS_BYTE_MESSAGE 1 Channel.SEND_OPTIONS_SECURE 16 Channel.SEND_OPTIONS_SYNCHRONIZED_ACK 4 Channel.SEND_OPTIONS_USE_ACK 2	11 (async with ack)

The channelSendOptions value is used by the DeltaManager only. If you switch to the BackupManager, this options value is not used. The default value of 11, asynchronous sent with ACK, is adequate in most cases.

Configuring DeltaManager or BackupManager

The <Manager> component is the first mandatory nested component within the <Cluster>. Here, you can configure an instance of either DeltaManager or BackupManager using the following attributes.

Attribute Name	Description	Default
className	For SimpleTcpCluster, only org.apache.catalina.ha.session.DeltaManager or org.apache.catalina.ha.session.BackupManager are available.	
name	A name for the cluster manager. The same name should be used on all instances.	
notifyListeners-OnReplication	Indicates if any session listeners should be notified when sessions are replicated between Tomcat instances.	false
expireSessions-OnShutdown	Specifies whether it is necessary to expire (and replicate the expiry) of all sessions upon application shutdown.	false
domainReplication	Specifies whether replication should be limited to domain members only; the domain is set as an attribute of the <Membership> tag. This option is available only with the DeltaManager.	false
mapSendOptions	When using the BackupManager, this maps the send options that are set to trigger interceptors. See the channelSendOptions attribute of the <Cluster> tag for possible values.	8 (async)

Both DeltaManager and BackupManager send replication information to others via channels from the Apache Tribes group communications library.

Configuring Apache Tribes Communications Framework

A channel is an abstract endpoint, analogous to a socket, that a member of the group can send and receive replicated information through. Channels are managed and implemented by the Apache Tribes communications framework. The Apache Tribes framework provides the following services:

- Membership service based on multicast heartbeat and TCP failure detection; dynamically determines the members in a group.
- Reliable messaging service between the different members of a group, providing a range of message delivery guarantees.

You can configure the exact characteristic of the channel used via:

- Configuration of the `<Membership>` subelement
- Configuration of the `<Receiver>` and `<Sender>` (and associated `<Transport>`) subelements
- Configuration of `<Interceptor>` subelements

Thorough coverage of the Apache Tribes communications framework is beyond the scope of this chapter; interested readers should peruse the detailed Apache Tribes documentation included with the Tomcat 6 distribution (as part of the `webapps/docs` application). The following subsections explain the specific configuration used in this example.

The `<Channel>` element has only one attribute as illustrated in the table that follows.

Attribute Name	Description	Default
`className`	It must be `org.apache.catalina.tribes.group.GroupChannel`, the only supported implementation.	

Configuring the <Membership> Subelement

The `<Membership>` subelement configures the membership service. This service is based on sending a multicast heartbeat regularly, and you need to select the *frequency*, *dropTime*, and multicast address and port to be used. This service determines and maintains information on the machines that are considered part of the group (cluster) at any point in time. The following table lists some of the attributes on the `<Membership>` element.

Attribute Name	Description	Default
`className`	Currently, only `org.apache.catalina.tribes.membership.McastService` is available.	
`address`	The multicast address selected for this instance. You can use a different multicast address and port to partition your LAN into multiple groups (clusters). However, because of the network traffic that even one replicating cluster generates, this partitioning feature is not as useful as it seems.	228.0.0.4
`port`	The multicast port used. You can use a different address:port for multicast to partition your LAN into multiple groups.	45564

Table continued on following page

Attribute Name	Description	Default
`frequency`	Frequency with which the service sends out multicast heartbeats (in milliseconds).	500
`dropTime`	The time elapsed without heartbeats before the service considers a member has died, and removes it from the group (in milliseconds).	3000
`ttl`	Sets the time-to-live value for multicast messages sent. Typically, you need to adjust this only if you need to route your multicast heartbeat packets through routers (not recommended).	
`soTimeout`	The `SO_TIMEOUT` value on the socket that multicast messages are sent to. Controls the maximum time to wait for a send/receive to complete. The value of `0` sets `SO_TIMEOUT` the same as the frequency attribute in milliseconds.	0
`domain`	For partitioning group members into separate domains for replication. See also the `domainReplication` attribute of the `<Manager>` element.	
`bind`	The IP address of the adapter that the service should bind to. If your machine has multiple network adapters, you should set this to make sure multicast packets are sent on the correct network.	0.0.0.0

The possible multicast address range is from 224.0.0.1 to 239.255.255.255; stay away from 224.*x.x.x* as well as 239.*x.x.x* because they have special meaning with certain network routers and switches.

Configuring <Receiver> and <Sender> Subelements

The `<Receiver>` element configures the TCP receiver component of the Apache Tribes framework. This component receives the replicated data information from other members. The frequently used configurable parameters are shown in the following table:

Attribute Name	Description	Default
`className`	Either `org.apache.catalina.tribes.transport.nio.NioReceiver` or `org.apache.catalina.tribes.transport.bio.BioReceiver`. The non-blocking NIO version is recommended, but if you are using an older Java VM where NIO has too many bugs, the blocking BIO version may be your only choice.	
`address`	The IP to bind to receive incoming TCP data. You should definitely set this on multi-homed machines in order to select a specific adapter to listen for incoming data. The value of `auto` tells the Java VM to take the first adapter returned from the system socket call — this works when you have one adapter on your machine.	auto
`port`	Selects the port to use for incoming TCP data. You can tell the framework to hunt for an available port by setting the `autoBind` attribute.	4000

Attribute Name	Description	Default
autoBind	Tells the framework to hunt for an available port, starting from the port number specified in the port attribute. A value of 1000 tells the framework to hunt for up to 1000 ports.	1000
selectorTimeout	Bypass for old NIO bug. Set the milliseconds timeout while polling for incoming messages.	5000
maxThreads	The maximum number of threads to create to receive incoming messages. This should be set to the same number of members for small clusters.	6
minThreads	The minimum number of threads to create when receiving incoming messages.	6

The <Sender> element configures the sender component of the Apache Tribes framework. This component uses TCP to send replicated data to other members of the group. This element has only one attribute, as shown in the table that follows.

Attribute Name	Description	Default
className	Only org.apache.catalina.tribes.transport.ReplicationTransmitter is available at this time.	

The <Sender> element must contain a <Transport> subelement. The <Transport> element has the frequently used attributes that are listed in the table that follows.

Attribute Name	Description	Default
className	Either org.apache.catalina.tribes.transport.nio.PooledParallelSender or org.apache.catalina.tribes.transport.bio.PooledMultiSender. The non-blocking NIO version is recommended; but if you are using an older Java VM where NIO has too many bugs, the blocking BIO version may be your only choice.	
maxRetryAttempts	The number of retries the framework conducts when encountering socket-level errors during sending of a message.	1
timeout	The SO_TIMEOUT value on the socket that messages are sent on. Controls the maximum time to wait for a send to complete. The value of 0 sets SO_TIMEOUT to the same as the frequency attribute in milliseconds	3000
poolsize	Available only when the non-blocking org.apache.catalina.tribes.transport.nio.PooledParallelSender sender component is selected. This controls the maximum number of TCP connections opened by the sender between the current and another member in the group.	25

Configuring <Interceptor> Subelements

Interceptor components are nested elements of `<Channel>` and are message-processing components that can be flexibly chained together to alter the behavior or add value to the operation of a channel.

Most `<Interceptor>` subelements have an `optionFlag` attribute that can control which send option flag will trigger its operation (allowing selective processing on a per-message basis). However, at the present time, this option flag is mostly hard-coded into the interceptor component itself. The most frequently used interceptors are tabulated in the following table. You can expect improvement on the existing interceptors, and more interceptors to become available in the near future.

Attribute Name	Description
`org.apache.catalina.tribes.group.interceptors.TcpFailureDetector`	When membership pings do not arrive in time, or TCP failure is reported, this interceptor attempts to connect to the problematic member to validate that the member is no longer reachable before the membership list is adjusted.
`org.apache.catalina.tribes.group.interceptors.MessageDispatch15Interceptor`	The asynchronous message dispatcher; triggered by default send option value 8. This is hard-coded in Apache Tribes currently.
`org.apache.catalina.tribes.group.interceptors.ThroughputInterceptor`	Logs cluster message throughput information (at INFO log level) to the Tomcat log.

A Replication <Valve> Element

Another component essential for in-memory replication is a replication `<Valve>` element. This element acts as a filter. The filter reduces the actual session replication network traffic by determining if the current session needs to be replicated at the end of the request processing cycle. The filter attribute of the replication Valve may be used to override and stop session replication for specific matching requests. Even though the `<Valve>` element is configured inside the `<Cluster>` element, structurally it is considered to be within the element containing the `<Cluster>` element (in this case, the `<Valve>` is considered to be in the `<Host>` element).

For this example, the replication `<Valve>` element is configured as follows:

```
          <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"
        filter=".*\.gif;.*\.js;.*\.jpg;.*\.htm;.*\.html;.*\.txt;"/>
```

The filter setting here filters out any requests for static pages, graphic pages, or JavaScript pages. These pages do not modify the session values. The following table shows the frequently used attributes of the replication Valve.

Attribute	Description
`className`	The Valve's implementation class. It must be `org.apache.catalina.ha.tcp.ReplicationValve`.

Attribute	Description
`filter`	A semicolon-delimited list of URL patterns for requests that are to be filtered out (i.e., excluded for session replication).

The JvmRouteBinderValve

This is a specialized valve that frequently needs to be configured when `mod_jk` is used in the `org.apache.catalina.ha.session.JvmRouteBinderValve`. This specialized valve works with `mod_jk`. This valve ensures session stickiness during a fail-over. Once a session is failed-over from machine1 to machine2, this valve ensures the session is sticky to machine 2 (when the valve is configured and enabled). This work is done by changing the `jvmWorker` associated with the session ID when a fail-over occurs.

The Farm Deployer

The `<Deployer>` subelement of a `<Cluster>` enables the deployment, reinstallation, and uninstallation of Web applications (WARs) to a cluster of Tomcat instances as a target. This simplifies the often tedious task of installing the same Web application to every instance of a cluster.

At the time of this writing, this feature is a work-in-progress, and not yet in functional order. Future releases of Tomcat 6 may support farm-based deployment.

The Cluster Listeners

Some of the work of the cluster is performed by hooking up listeners to replication messages that are passing through it. Currently, most of the configurable listeners are associated with other components and must be configured when the associated component is configured.

For example, you must configure `org.apache.catalina.ha.session.JvmRouteSessionIDBinderListener` if you are using the `JvmRouteBinderValve` to ensure session stickiness transfers with a fail-over.

Also you must configure `org.apache.catalina.ha.session.ClusterSessionListener` if you are using the `DeltaManager` because this listener forwards the messages to the manager for delta and merging operations.

More cluster listeners may become available with future releases of Tomcat 6.

Setting Up the Test JSP for Tomcat Session Replication

The JSP used in the testing here is named `sesstest.jsp`.

```
<%@page language="java" %>
<html>
<body>
<h1><font color="red">Session serviced by machine1</font></h1>
<table aligh="center" border="1">
<tr>
<td>
  Session ID</td>
<td><%= session.getId() %></td>
<% session.setAttribute("abc","abc"); %>
```

(continued)

```
  </tr>
  <tr>
  <td>
    Created on</td>
    <td><%= session.getCreationTime() %></td>
  </tr>
  </table>
  </body>
  </html>
```

Note that this is very similar to the JSP used in Chapter 11. It simply gets the session ID and date and displays them. The `setAttribute()` method is also called explicitly to cause a change in the session, triggering the replication mechanism to be tested. When accessing this JSP across server instances, a matching session ID indicates that a session has migrated/replicated from server to server in the cluster. In addition, each server has a JSP that displays the heading in a different color. The following code examples show the single-line difference for each copy of `sesstest.jsp`.

For machine1, the line would be

```
<h1><font color="red">Session serviced by machine1</font></h1>
```

For machine2, the line would be

```
<h1><font color="green">Session serviced by machine2</font></h1>
```

For machine3, the line would be

```
<h1><font color="blue">Session serviced by machine3</font></h1>
```

The `sesstest.jsp` is placed into the `%CATALINA_BASE%/webapps/jsp-examples` directory of each of the Tomcat instances in the cluster. For example, the `sesstest.jsp` for machine1 is placed into the `\cluster\machine1\webapps\jsp-examples` directory. This enables simple access to the test JSP without the need to configure another Web application.

Testing Tomcat 6's In-Memory Session Replication Cluster

To test the in-memory session replication cluster, perform the following steps:

1. In the `server.xml` files of the three instances, ensure that the `<Cluster>` and the replication `<Valve>` elements are uncommented, and that the `<Manager>` element (for later examples) is commented out. In addition, ensure that the `<Context>` element is commented out, as this is used only for the Persistent Session Manager example later.
2. Start the three configured Tomcat 6 instances, with the batch files in the `<Tomcat installation Directory>/bin` directory (start1, start2, and start3); be sure to wait sufficient time until one fully starts up before starting another. (The replication manager takes a little longer than normal to start up.)
3. Start the Apache server with the `mod_jk` module.
4. Start an instance of a browser and point it to the following URL:

   `http://<your ip address>/examples/jsp/sesstest.jsp`

Initially, your display should be similar to what is shown in Figure 17-11. This indicates that machine1 is servicing your incoming request, and a session is created with the ID as displayed.

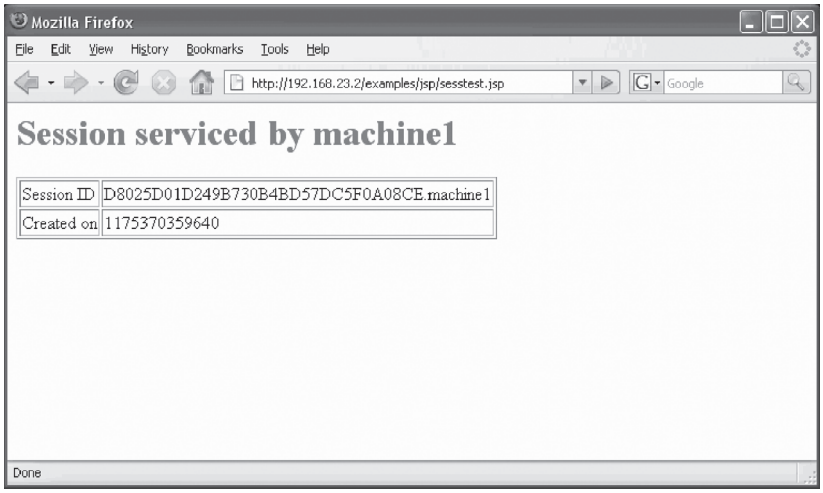

Figure 17-11: Establishing a session on machine1 in the cluster

You can click Reload a few times, and you can see that every request is directed to machine1 — with the same session ID. This is because sticky sessions (sometimes called seamless sessions) are enabled for `mod_jk`. This is almost always desirable because the machine currently working on the session is the one that is certainly ready to continue working on it. There is no need to use replicated session information if there is no failure.

Observing HA Fail-Over

To see fail-over at work, note both the machine that is servicing your request and the session ID. Now, go to the console of that Tomcat instance and terminate it (press Ctrl+C or close the console window). This represents a sudden failure. Wait a little while (30 seconds or so) and then click Reload. Notice that your session is now handled by one of the remaining servers, with no change in the session ID. This is shown in Figure 17-12.

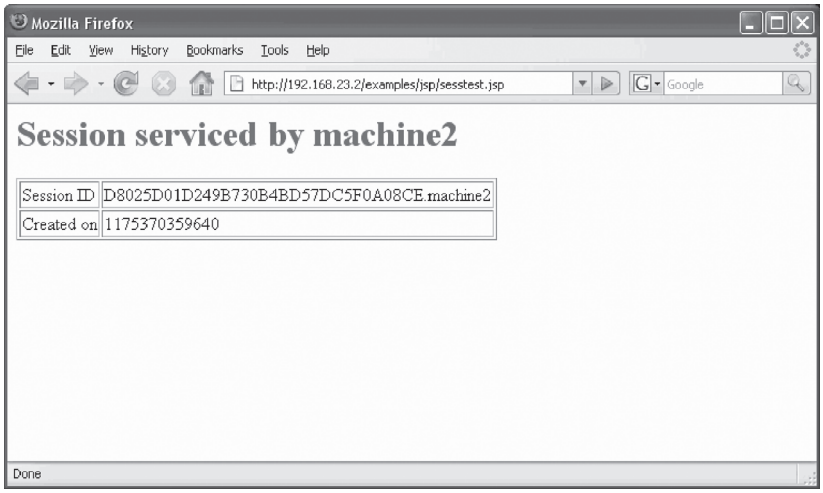

Figure 17-12: Failed-over session serviced by machine2 — same session ID

If you have the means to observe network traffic over the cluster LAN (i.e, a hardware Ethernet monitor) you should see significant traffic even with this simple example. Devising a load test (as described in Chapter 20) enables you to determine if the typical incoming request rate will be adequately served by this clustering backend.

> *As mentioned previously, you can substitute the BackupManager for the DeltaManager if you want to reduce traffic on the cluster LAN. However, at the time of this writing, BackupManager is still a work-in-progress.*

Observing Load Balancing for Requests

To see round-robin work distribution or load balancing at work, you need to open another instance of the browser and try the same URL again:

```
http://<your ip address>/examples/jsp/sesstest.jsp
```

Note that the new session has a new session ID and is serviced by the next-in-line Tomcat session instance. This new session is served by the same server instance as long as there are no further failures.

You should try opening more browser instances and accessing the same URL. Each of the browser instances should have its own unique session ID, and will be serviced in a round-robin fashion by the three Tomcat instances in the cluster.

Note that when using Firefox, you must close down any existing session before this works.

This completes the exploration of memory session replication. Next, we cover an alternative backend using a Persistent Session Manager.

Backend 2: Persistent Session Manager with a Shared File Store

This section describes the configuration for a Persistent Session Manager. This Persistent Session Manager uses a shared directory to store its persistent session.

To configure the cluster to work with a Persistent Session Manager, first comment out the `<Cluster>` elements in each of the `server.xml` files. This disables the in-memory replication mechanism.

Configuring the <Manager> Element

Next, add a `context.xml` file to each of the clustered nodes under the `webapps/examples/META-INF` directory (if this directory does not exist, create it). If you are using the code distribution, the following `context.xml` is already in place, but may already be commented out:

```
<Context>
  <Manager className="org.apache.catalina.session.PersistentManager" >
     <Store className="org.apache.catalina.session.FileStore"
      directory="c:\\cluster\\shareddir" />
  </Manager>
</Context>
```

You can change the `c:\cluster\shareddir` directory to any other directory that you are using to store shared session information. Just make sure that you change the directory attribute of the `<Valve>` element.

Unlike the previous example, note that the context for the `jsp-examples` Web application is explicitly specified here (instead of letting Tomcat 6 create a default one). This is because the Persistent Session Manager must be configured in the form of a nested `<Manager>` element. The `<Manager>` element can reside only within a `<Context>` element to persist the sessions created within that particular Web application. See Chapter 6 for more details on the specific allowed attributes for the `<Manager>` element.

The <Store> Nested Element

The only allowed subelement of a `<Manager>` element is a `<Store>` element. Only two different stores are available with Tomcat 6: One uses a shared file directory to store the sessions, and the other one uses an RDBMS through JDBC to store the sessions. In this case, the Tomcat instance is configured with the shared file system store.

Note that the exact mechanism to share the specified directory is system- and installation-specific. In this case, the shared directory is actually the same physical directory because all the Tomcat instances executed reside on the same physical machine. In production systems, where each Tomcat instance runs on a different physical machine in the cluster, the directory may be shared by operating system–specific means (NFS on Linux and Solaris, SMB on Win32 servers, and so on). Of course, you should make sure that the shared directory specified indeed exists and is accessible.

Testing a Shared File System–Based Persistent Session Cluster

To test and see the level of HA provided by this solution, first start the three Tomcat instances using the start1, start2, and start3 batch files in the `<Tomcat Installation Directory>/bin` directory. Next, start the Apache server.

Start a browser instance and try accessing the following URL:

```
http://<ip address of your host>/examples/jsp/sesstest.jsp
```

You should see machine1 servicing your request, and the session ID displayed. Write down this session ID. Click Reload a few times, and notice that machine1 continues to service your request and that the session ID remains identical. This is sticky session working as desired.

Open another browser instance and access the same URL. This time, machine2 will service the request (thanks to round-robin load balancing). If you reload the page, machine2 will continue to service the request — again sticky session is working as configured.

Observing an Orderly Fail-Over

To observe HA at work, try to simulate a fault by machine1. Go to the `<Tomcat installation directory>/bin` directory and execute the following batch file:

```
C:\>  stop1
```

This shuts down the machine1 instance. Because the machine1 instance is no longer running, if you click Reload on the browser serviced by machine1, it will be round-robin load-balanced to machine3

(machine2 serviced an earlier request from another browser instance). Note that even though machine3 is now handling the request, the session ID stays the same. In other words, the application has successfully failed over from machine1 to machine3 and in a manner transparent to the user.

This works mainly because an orderly shutdown was performed. During the orderly shutdown, all of the sessions in the Persistent Session Manager were persisted to the store (shared directory). This may be useful in many situations. For example, a server can be taken out of service for maintenance and upgrades without affecting the ongoing cluster operation. In some cases, it may also be possible for a fault-detection mechanism to detect a problem and perform an orderly shutdown (hardware-uninterruptible power backup sources often offer this feature).

The picture is slightly different, however, when the server instance shutdown is not orderly.

Observing a Sudden Fail-Over

The key limitation to remember when working with a Persistent Session Manager shared-store backend is that any sessions or changes not persisted at crash time are lost.

To observe this limitation, start a new browser instance and point it to the following URL:

```
http://<ip address of your host>/examples/jsp/sesstest.jsp
```

Note the session ID. (Don't use any old browser instance because an older session may already be persisted.)

Also note the Tomcat instance that is servicing this request/session. Now, go to the console running the Tomcat instance and press Ctrl+C or close the window.

This simulates a sudden system crash shortly after the session is created. Next, click your browser's Reload button. Note that a new instance is now servicing the request, but the session ID you observe may be either of the following:

- **The same session ID:** In this case, the fail-over was successful because the session was persisted before the crash.
- **A brand-new session ID:** In this case, the session was lost during fail-over because it was not persisted before the crash.

If you repeat this test a few times, you are likely to see both behaviors. It is possible to tweak the attributes of a Persistent Session Manager or the store to alter the time between session creation or modification and session persistence. However, the Persistent Session Manager is not designed with sharing sessions across clustered machines in mind, and it does not provide a hard guarantee for the time window between creation/update and persistence.

Despite the small potential of lost sessions, this level of HA support (which improved over the scenario with no session sharing at all) can be adequate for many real-world deployments. Its simplicity of configuration and inexpensive implementation are two advantages that should not be overlooked.

The next section takes a look at an alternative store mechanism for the Persistent Session Manager.

Backend 3: Persistent Session Manager with a JDBC Store

Instead of sharing a directory between the Tomcat instances, it is also possible to write all shared session information into an RDBMS through JDBC. This is done through the same Persistent Session Manager, but using a JDBC `<Store>` element instead of a shared file system–based `<Store>` element.

A few scenarios in which you may want to use the JDBC store instead of a shared file-system store are as follows:

- When a shared file system is not available to all the physical machines in the cluster, but a common JDBC connection is
- When the JDBC connection and RDBMS offer higher performance or provide the robustness guarantee that you may need
- When the JDBC connection to the shared RDBMS is made through a separate physical hardware connection (another LAN, Firewire, proprietary communication link, and so on) that is less prone to failure (that is, it may be redundantly implemented through an RDBMS-level cluster)
- When the shared file-system access traffic is over the same LAN that is handling the routed `mod_jk` requests, and the JDBC connection is made over a separate LAN or communications link
- When normal operations of the cluster involve a large number of sessions that may be swapped out at any moment in time

In this example, a MySQL RDBMS table is used to store session information and enable a cluster of Tomcat servers to share session information via the Persistent Session Manager, through JDBC.

All the frontend configurations and `context.xml` modifications made in the previous example continue to work. All that you need to do is replace the previous shared file system–based `<Store>` with the JDBC-based `<Store>` subelement within each of the `<Manager>` elements:

```
        <Store className="org.apache.catalina.session.JDBCStore"
   connectionURL="jdbc:mysql://localhost/wroxtomcat?user=empro&amp;password=empass"
         driverName="com.mysql.jdbc.Driver"
         sessionIdCol="session_id"
         sessionValidCol="valid_session"
         sessionMaxInactiveCol="max_inactive"
         sessionLastAccessedCol="last_access"
         sessionTable = "tomcat_sessions"
         sessionAppCol = "app_context"
         sessionDataCol = "session_data"
        />
```

The user and password information is part of the `connectionURL`. Because the syntax of the URL requires the ampersand (`&`), it must be escaped within this XML-based configuration file as `&`.

The creation of the corresponding MySQL table called `tomcat_sessions` is shown later. The following table describes the attributes supported by the JDBC `<Store>` element.

Attribute Name	Description	Default
`className`	The JDBC Store implementation, which must be `org.apache.catalina.session.JDBCStore`.	
`checkInterval`	Time between scans of persisted sessions for any expired session. Specified in seconds.	60
`connectionURL`	The JDBC URL used to connect to the database instance. Note that user and password must also be part of the URL, as there are no corresponding user or password attributes.	
`driverName`	The Java language class name for the JDBC driver to use.	
`sessionTable`	The name of the RDBMS table that is used to store session information.	`tomcat$sessions`
`sessionIdCol`	The name of the database table column that contains the session ID information. This column should be the primary key of the table because it is the key used for lookup most frequently.	`id`
`sessionValidCol`	The name of a database table column that contains a flag indicating if the associated session (row) is still valid.	`valid`
`sessionMaxInactiveCol`	The name of the database table column used to persist the value of the `MaxInactiveInterval` property for the session.	`maxinactive`
`sessionLastAccessedCol`	The name of the database table column used to persist the value of the `lastAccessedTime` property for the session.	`lastaccess`
`sessionAppCol`	The name of the database table column used to persist the Engine, Host, and Context information for the session.	`app`
`sessionDataCol`	The name of the database table column used to persist the actual session data (serialized session attributes).	`data`

You need the MySQL driver for Java, called MySQL Connector/J, installed. Make sure you have copied MySQL Connector/J into Tomcat's `lib` *directory; otherwise, the Persistent Session Manager may fail with a* `NullPointerException` *or silently. If you have the examples shown in Chapter 13 working, you are all set.*

To create the `tomcat_sessions` table, the following SQL script can be used. This script is found in the `mksesstbl.sql` file within the code distribution:

```
USE wroxtomcat;
drop table if exists tomcat_sessions;
create table tomcat_sessions (
  session_id     varchar(100) not null primary key,
  valid_session  char(1) not null,
  max_inactive   int not null,
  last_access    bigint not null,
  app_context       varchar(255),
  session_data   mediumblob,
  KEY kapp_context(app_context)
);
```

An additional complication can arise during configuration when multiple instances of Tomcat run on the same physical machine. This is because the combination of host name and user is not unique during RDBMS access. Therefore, it is necessary to create three different users for machine1, machine2, and machine3 access, respectively. The following MySQL commands create these additional users (provided you have the necessary administrator privilege on the MySQL server):

```
GRANT SELECT,INSERT,UPDATE,DELETE ON wroxtomcat.tomcat_sessions TO
'empro'@'localhost' IDENTIFIED BY 'empass';
GRANT SELECT,INSERT,UPDATE,DELETE ON wroxtomcat.tomcat_sessions TO
'empro1'@'localhost' IDENTIFIED BY 'empass';
GRANT SELECT,INSERT,UPDATE,DELETE ON wroxtomcat.tomcat_sessions TO
'empro2'@'localhost' IDENTIFIED BY 'empass';
```

These commands create the users and passwords shown in the following table for concurrent access of the `tomcat_sessions` table.

Tomcat Instance	User	Password
machine1	empro	empass
machine2	empro1	empass
machine3	empro2	empass

If you examine the actual `mksesstbl.sql` file in the code distribution, you will also see the SQL grant statements for creating the `empro`, `empro1`, and `empro2` JDBC access accounts. This saves you from entering the commands manually.

The following command line executes the SQL script (assuming that you have a create table privilege on the database):

```
$ mysql < mksesstbl.sql
$ mysqladmin -u???? -p????? flush-privileges
```

The second statement is needed to flush the privileges cache and make the `grant` statements effective immediately.

After the table creation script is executed, you are ready to test the JDBC-based Persistent Session Manager backend.

Testing a Tomcat Cluster with JDBC Persistent Session Manager Backend

The steps for testing the Tomcat cluster with a JDBC Persistent Session Manager backend are exactly the same as those for the shared file system Persistent Session Manager. See the instructions for the preceding example for more details. There should be no observable difference between the behaviors of the two examples. A JDBC-based backend provides a robust store mechanism and is a potentially higher performance mechanism when numerous sessions need to be persisted/shared.

The Complexity of Clustering

While the setup and configuration of Tomcat 6 clustering can be a daunting task, the capability to obtain tangible benefits from a clustered configuration is the most complex. Several misconceptions (sometimes called urban legends) prevail:

- ❏ Performance of a Web application increases with clusters.
- ❏ Response time to the user improves (decreases) with clusters.
- ❏ Clustering is the most inexpensive way to solve performance and response-time problems observed with Web applications.

Clustering and Performance

The word "performance" means different things to different people. Often, it is not politically correct or survival savvy to correct marketing gurus or system architects on this point, but to state a general "performance improvement" objective without specific metrics is akin to saying that computers will solve all our problems.

Many aspects of what is perceived as performance improvement by the end user cannot be achieved by a clustering solution. The scenario that most naturally lends itself to a clustering solution, and the one that clearly benefits from such a configuration, derives from the following:

- ❏ A Web application running on a single Tomcat instance on one physical machine handles all incoming requests with no problem during normal incoming load volume.
- ❏ The machine starts to slow down or fail under heavy incoming traffic volume.
- ❏ Most important, analysis of the failure/slowdown reveals that the bottleneck is in CPU saturation while processing Tomcat servlets/JSPs.

It is of utmost importance to reiterate the last point.

> *Until an observed performance slowdown caused by incoming load on a single machine can be isolated to the single factor of CPU saturation resulting from Tomcat-hosted application processing, the benefits of a Tomcat 6 clustering configuration cannot be ascertained.*

For example, no amount of expensive hardware or sophisticated configuration spent on Tomcat clustering will solve system bottleneck problems that pertain to faulty non-scalable application logic, network bandwidth saturation, RDBMS access, and so on.

Obviously, a million other variations of real-world situations that do not quite fit this scenario are possible, and your success with applying Tomcat clustering to the problem will vary accordingly.

Clustering and Response Time

In general, the addition of clustering may actually increase observed individual response time, rather than reduce it, especially when contrasted with a lightly loaded single-server solution. This is because of the overhead involved in frontend load balancing and session replication. Therefore, one cannot guarantee a user that he or she will observe improved response time when using a clustered solution (versus a single-server solution).

The single-performance metric that should improve with a properly configured and applied Tomcat 6 cluster is *system throughput* (measured in requests processed per time unit) under maximum load. This means that the clustered system as a whole should handle more requests without failing. This is the basic premise of horizontal scalability (*scaling out*).

Solving Performance Problems with Clustering

It should be clear by this point that clustering should not be used as "magic dust" to solve performance problems. Throwing more hardware at the problem may not make it go away. Instead, a proper analysis of the system, including isolation of the bottleneck element, must be performed before applying clustering as a solution. In many cases, the analysis reveals other factors that cannot benefit from a clustering solution.

When a system performance bottleneck can be isolated to the saturation of computing resources related to Tomcat application processing, horizontal scaling can be achieved via Tomcat 6 clustering configuration.

Of course, if you have other non-performance–related system design goals that can benefit from distributed replicated logic (such as high availability for certain Web applications), clustering may still provide benefits.

Summary

This chapter covered the following key points:

- Widespread production deployments of Tomcat servers have motivated the Tomcat development team to refactor the server for performance and give serious consideration to real-world deployment issues, including scalability and high availability.
- Originally exclusive to proprietary hardware solutions, scalable and highly available clusters can now be achieved inexpensively using Tomcat 6 servers running on PCs, and commodity networking hardware.
- Tomcat 6 supports clustering right out of the box. A variety of mix-and-match components are available, enabling the administrator to select the most appropriate configuration for the situation at hand.
- A clustering Web-tier server solution consists of a load-balancing frontend and a state/session sharing backend, with the cluster of Tomcat 6 servers sandwiched in between.

- You have many options for a load balancing frontend solution, but the Apache server with the `mod_jk` (or `mod_proxy`) is the technology of choice for production use because of the popularity and performance of Apache. This solution supports a round-robin distributor that respects a specified load factor, and also supports sticky sessions.
- For the session-sharing backend technology, Tomcat 6 comes with support for group communications based on the memory-session replication mechanism. When enabled, multicast packets are used to maintain cluster membership, while TCP connections are made between servers to share session information. This configuration enables any server in the cluster to service any session, providing a highly available clustering solution. However, this solution should be deployed with care because of its escalating network bandwidth requirement as cluster size and session replication traffic increase.
- The Persistent Session Manager built into Tomcat can also be used for a form of session sharing. Either a file-based store or a JDBC store can be configured. Once a shared file system or RDBMS is configured, any persisted session becomes available to all the server instances. When configured together with a frontend that supports sticky sessions, this solution can be a very effective high-availability solution that minimizes lost sessions during a fail-over.

In summary, Tomcat 6 provides the administrator with a rich toolbox of components and mechanisms at various layers of the server architecture, to design and build functional server clusters. These cluster mechanisms can fulfill most scalability and/or high availability requirements in today's production environments.

Chapter 18 discusses embedded Tomcat.

18

Embedded Tomcat

Ever since the initial availability of the Tomcat server, some developers have wanted or needed to create applications that have full control over the server's lifecycle and internal operation. When the entire Tomcat server is contained within a custom application, it is said to be operating in *embedded mode*. While provisions were made for embedding Tomcat into applications in past releases (as far back as Tomcat 3.*x*), these older provisions were rather ad hoc and problem-prone because the earliest versions of Tomcat did not account for the embedded mode of operation. Tomcat 5 changed this landscape completely. Embedded operation is an explicitly supported mode, and Tomcat 6 and future Tomcat designs will evolve to satisfy the emerging requirements from embedded users.

This chapter explores the embedded mode of Tomcat 6 in the following areas:

- Why embedding Tomcat may be important for many projects
- Programmatically embedding Tomcat
- Developing a real embedded Tomcat application

Hands-on examples are provided so that you can experiment with configuring embedded Tomcat instances. This chapter focuses more on the under-the-hood aspects of embedded Tomcat implementations.

By the end of this chapter, you should have a comprehensive understanding of why you might use embedded mode and how to operate Tomcat in embedded mode. You will be able to facilitate the creation of applications and systems that operate embedded Tomcat instances.

> *Tomcat versions before 6.x included Ant scripts that provided embedded control and access through JMX. Although Tomcat still includes JMX extensions, it has dropped support of the embedded Ant scripts, and therefore this chapter does not cover them. This chapter covers the programmatic embedded implementation.*

Importance of Embedded Tomcat in Modern System Design

Tomcat 6 is a container for JSPs and servlets. JSPs and servlets are componentized building blocks for Web applications or services. These components (along with custom Java classes and other Web resources) are managed and processed by the Tomcat container. The componentized nature of the applications makes them easy to construct, deploy, and maintain.

One specific scenario for such a use of Tomcat involves Web applications that are deployed and undeployed dynamically on one or more general-purpose Tomcat containers. In some situations, however, the application running within the Tomcat server may not need to change, or may need to change infrequently.

One example might be the use of Tomcat to create a Web interface for the configuration of a piece of hardware equipment. Because the hardware equipment will not change, you need to modify the configuration application logic infrequently. Another example might be the display and management of a standalone (non-networked) interactive catalog from a CD-ROM. The items and prices in the catalog are fixed once the CD-ROM is printed.

In these situations, the application that is running under Tomcat may be considered to be an embedded application. While it is certainly possible to start the generalized standalone Tomcat server for running such an application, it makes more design sense to minimize the memory footprint and CPU utilization by running "just enough" Tomcat to support the application. Embedded Tomcat 5 provides this flexibility to run "just enough" Tomcat.

Figure 18-1 illustrates a custom hardware scenario that may require embedded Tomcat.

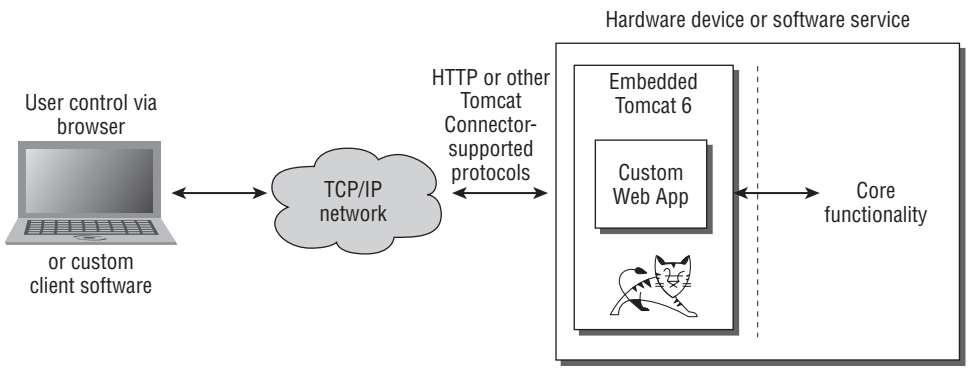

Figure 18-1: Embedded Tomcat 6 providing custom network facade for device/service

In Figure 18-1, notice that a componentized Web application actually resides within the Tomcat container. This Web application provides the user with control over the device/service, as well as live status information. The servlets and JSPs communicate directly with the internal logic. The Tomcat container, in turn, is embedded as part of the device or service.

Typical Embedded Application Scenarios

Traditionally, most embedded applications are created from scratch as a monolithic code module, potentially utilizing some proprietary or third-party software libraries. Applications that require Web-based networking are especially difficult to create because of the complexity of HTTP protocol handling and HTML generation. By embedding the Tomcat server, application creation is simplified, as all of this complexity is handled by Tomcat.

To device/service designers and developers, the appealing benefits of embedding a Tomcat server within an application include the following:

- Rapid development time and a shorter project cycle
- Large developer pool with training on Java, servlets, and JSP
- Wide availability of supporting tools for servlet and JSP coding and testing
- Ease of coding, testing, and maintaining application logic
- Getting an HTTP server and Web interface for "free"

Some typical scenarios in which embedding Tomcat in an application may make sense include the following:

- Providing a Web interface front-end for a complex hardware and/or software system
- Creating a standalone "turnkey network appliance" box that hosts a Web-based application (sometimes called *network black-boxes* or *network appliances*)
- Creating a complex customized server in which serving HTML pages and/or servlets and/or JSP is a part of the required application logic
- Creating a standalone, fat-client version of an existing Web application, whereby the application code is already available and tested
- Providing common operation/management code that can be scripted across multiple operating systems and hardware platforms

Apache Geronimo is a great example of a very complex server using embedded Tomcat to make its own Web container. The embedded Tomcat enabled Geronimo to form a cohesive Web container with its own style of configuration files, known as GBeans. Embedding Tomcat allowed the Geronimo developers to integrate transactions, JNDI, and Java Enterprise Edition (Java EE) security right into the container. It gave the appearance that Geronimo had its very own Web container, while leveraging the best aspects of Tomcat.

> *Note that you must abide by the Apache Software License (see the* `LICENSE` *file in the top-level directory of the Tomcat Embedded Distribution) if your software product is being delivered with Tomcat 6 inside. This license is quite liberal but must be followed carefully. In a production or vendor shop, this will typically be the responsibility of the marketing, production, and legal staff.*

Developing with Embedded Tomcat

Tomcat itself is based on the embedded API. In fact, the main class that is used to launch Tomcat, Catalina, extends the `Embedded` object. When Tomcat launches, it uses the Apache Commons Digester to

read the contents of a few XML files, such as `service.xml`, to dynamically create the key Tomcat objects, such as the `Connector`, `Engine`, `Host`, and `Context`. The `Catalina` object puts these objects together as well as glues and wires in other items to make a fully working and specification-compliant Web container. Hence, the Catalina container itself is really an intricate and sophisticated use of embedded Tomcat.

Chapter 4 explained the relationships between the key Tomcat objects and their parent/child hierarchies. When Catalina launches, it uses the Digester to read the `service.xml` file and create the objects on-the-fly and enforce the relationships. When you use embedded Tomcat, you are able to create these objects programmatically, use the Digester, or write your own handler to read a customized configuration file to create the Tomcat objects. As stated earlier in the chapter, Apache Geronimo used embedded Tomcat as its main Web container and wrote its own configurations based on a GBean plan, which looks significantly different from the well-known `service.xml` file.

Most commonly, when embedding Tomcat in your applications, you want to provide a simple Web server inside your application, or to provide a very lightweight Java-based Web application. You may need to offer documents, images, or a simple management interface to control your application, or you may use it to allow HTTP-based services.

Programmatically Embedding Tomcat

Embedding Tomcat is relatively simple. Just as the Digester does with `service.xml`, you need to create the Tomcat object hierarchy and wire these objects together through their parent/child relationships. To embed Tomcat, you typically implement the following steps:

1. Create an `Embedded` object.
2. Create an `Engine` object and give it a name and tell it the name of the default `Host`.
3. Create one or more `Host` objects and give them each a name and a location to a directory that contains each host's Web applications. At least one of the `Host` objects should be named the same as the default in the `Engine`. Typically you create only one `Host` object when you are embedding Tomcat.
4. Add each `Host` as a child to the `Engine`.
5. Create one or more `Context` objects, which represent a Web application, set the parent class loader, and tell it which directory will hold your Web application. Typically, you will create one `Context` when you are embedding Tomcat, but it would not be unusual to have an embedded application serve multiple contexts.
6. Add each `Context` as a child to its respective `Host`.
7. If you are not using a default `web.xml` file for the embedded container, optionally create `Wrapper` objects (servlets) to handle the default servlet and JSPs, and set MIME types and default welcome files. This would be done only if you need to process default servlet calls or JSPs.
8. Add the `Wrapper` object(s) to the `Context` as children.
9. Create a `Connector` object to listen for HTTP requests.
10. Set the `Connector` in the `Embedded` object.
11. Start the embedded container by calling the `Embedded` object's `startServer()` method.

Tomcat comes with a default `web.xml` file that lives in the `conf` directory, which contains items that become a part of every web application. This includes setting a default servlet and a servlet to handle JSPs. It also sets the MIME types as well as the welcome files. If you wish to use a `web.xml` file, your Catalina home directory (or the directory that you claim is the root of your application) should contain a `conf` directory and should have the same `web.xml` file as is found in Tomcat. However, because you are embedding an application, it probably makes sense to not include the default `web.xml` file because it's an additional extraneous directory.

Let's take a look at an example of an embedded Tomcat implementation. The following code shows an object called `MyWebServer` that instantiates an embedded Tomcat instance and wire in a single `Engine`, `Host`, and `Context`. This example also creates the default `Wrapper` objects because the code doesn't use a `web.xml` file, and implements a `ShutdownHook` to clean up after itself after it is halted by pressing Ctrl+C.

```
 1    import org.apache.catalina.startup.Embedded;
 2    import org.apache.catalina.connector.Connector;
 3    import org.apache.catalina.Engine;
 4    import org.apache.catalina.Host;
 5    import org.apache.catalina.Context;
 6    import org.apache.catalina.Wrapper;
 7
 8    import java.io.File;
 9
10    public class MyWebServer {
11
12        private static final String DEFAULT_ENGINE = "default";
13        private static final String DEFAULT_HOST = "localhost";
14        private static final String WEB_APPS_NAME = "myapps";
15        private static final String DOC_BASE = "docbase";
16
17        private Embedded embedded;
18        private String catalinaHome;
19
20        public MyWebServer() {
21            // Register a shutdown hook to do a clean shutdown
22            Runtime.getRuntime().addShutdownHook(
23                    new Thread() {
24                        public void run() {
25                            stopServer();
26                        }
27                    });
28        }
29
30        private void init() throws Exception {
31            File home = (new File(".")).getCanonicalFile();
32            catalinaHome = home.getAbsolutePath();
33
34            embedded = new Embedded();
35            embedded.setCatalinaHome(catalinaHome);
36
37            // Create an Engine
38            Engine engine = embedded.createEngine();
39            engine.setName(DEFAULT_ENGINE);
```

(continued)

```
40              engine.setDefaultHost(DEFAULT_HOST);
41              embedded.addEngine(engine);
42
43              // Create a Host
44              File webAppsLocation = new File(home, WEB_APPS_NAME);
45              Host host = embedded.createHost(
46                      DEFAULT_HOST, webAppsLocation.getAbsolutePath());
47              engine.addChild(host);
48
49
50              // Add the context
51              File docBase = new File(webAppsLocation, DOC_BASE);
52              Context context = createContext("", docBase.getAbsolutePath());
53              host.addChild(context);
54
55              // Create a connector that listens on all addresses
56              // on port 5050
57              Connector connector = embedded.createConnector(
58                      (String)null, 5050, false);
59
60              // Wire up the connector
61              embedded.addConnector(connector);
62
63          }
64
65          private Context createContext(String path, String docBase){
66              // Create a Context
67              Context context = embedded.createContext(path, docBase);
68              context.setParentClassLoader(this.getClass().getClassLoader());
69
70              // Create a default servlet
71              Wrapper servlet = context.createWrapper();
72              servlet.setName("default");
73              servlet.setServletClass(
74                      "org.apache.catalina.servlets.DefaultServlet");
75              servlet.setLoadOnStartup(1);
76              servlet.addInitParameter("debug", "0");
77              servlet.addInitParameter("listings", "false");
78              context.addChild(servlet);
79              context.addServletMapping("/", "default");
80
81              // Create a handler for jsps
82              Wrapper jspServlet = context.createWrapper();
83              jspServlet.setName("jsp");
84              jspServlet.setServletClass(
85                      "org.apache.jasper.servlet.JspServlet");
86              jspServlet.addInitParameter("fork", "false");
87              jspServlet.addInitParameter("xpoweredBy", "false");
88              jspServlet.setLoadOnStartup(2);
89              context.addChild(jspServlet);
90              context.addServletMapping("*.jsp", "jsp");
91              context.addServletMapping("*.jspx", "jsp");
92
93              // Set seme default welcome files
94              context.addWelcomeFile("index.html");
```

```
 95              context.addWelcomeFile("index.htm");
 96              context.addWelcomeFile("index.jsp");
 97              context.setSessionTimeout(30);
 98
 99              // Add some mime mappings
100              context.addMimeMapping("html", "text/html");
101              context.addMimeMapping("htm", "text/html");
102              context.addMimeMapping("gif", "image/gif");
103              context.addMimeMapping("jpg", "image/jpeg");
104              context.addMimeMapping("png", "image/png");
105              context.addMimeMapping("js", "text/javascript");
106              context.addMimeMapping("css", "text/css");
107              context.addMimeMapping("pdf", "application/pdf");
108
109              return context;
110          }
111
112          public void startServer() throws Exception {
113              init();
114              embedded.start();
115          }
116
117          public void stopServer() {
118              if (embedded != null) {
119                  try {
120                      System.out.println("Shutting down MyServer...");
121                      embedded.stop();
122                      System.out.println("MyServer shutdown.");
123                  } catch (Exception e) {
124                      //No need to do anything
125                  }
126              }
127          }
128
129          public static void main(String args[]) throws Exception {
130
131              MyWebServer server = new MyWebServer();
132              server.startServer();
133
134              // This code is just to prevent the sample
135              // application from terminating
136              synchronized (server) {
137                  server.wait();
138              }
139          }
140      }
```

Lines 20–28 show the constructor, and it sets up a `ShutdownHook` to properly shut down Tomcat. Although it may be a bit of an overkill for this example, it is a good practice to properly stop the Tomcat server with a `ShutdownHook`. The `ShutdownHook` is executed when the VM is about to shut down.

The `init()` method contains the lines that instantiate the `embedded` object and the Tomcat objects. Lines 31–32 determine the Catalina home location. This application assumes the Catalina home is the current directory from which you run the application.

```
34              embedded = new Embedded();
35              embedded.setCatalinaHome(catalinaHome);
```

The `Embedded` object is then used to create nearly all of the other Tomcat objects you are interested in. Lines 37–47 create an `Engine` object and the `engine`'s name and the default host are set. You then add the `engine` to the `Embedded` object.

```
37              // Create an Engine
38              Engine engine = embedded.createEngine();
39              engine.setName(DEFAULT_ENGINE);
40              engine.setDefaultHost(DEFAULT_HOST);
41              embedded.addEngine(engine);
```

After the `engine` has been created, you create the `Host` object, again from the `Embedded` object. When you create the `Host`, you must set the host's name and the path to its Web application's directory. The `Host` is then added as a child to the `Engine`.

```
43              // Create a Host
44              File webAppsLocation = new File(home, WEB_APPS_NAME);
45              Host host = embedded.createHost(
46                      DEFAULT_HOST, webAppsLocation.getAbsolutePath());
47              engine.addChild(host);
```

Finally, lines 50–53 create a `Context`, which is added as a child to the `Host`.

```
50              // Add the context
51              File docBase = new File(webAppsLocation, DOC_BASE);
52              Context context = createContext("", docBase.getAbsolutePath());
53              host.addChild(context);
```

In this instance, a local private method, `createContext()`, is called. This was done so that if you wanted to make several Web applications/contexts, you would need to pass only in the path that the container should answer to, and the `docbase` of where your application will reside.

```
67              Context context = embedded.createContext(path, docBase);
68              context.setParentClassLoader(this.getClass().getClassLoader());
```

Notice that line 52 calls the method with an empty path. This is the default context root, which means it answers to "`/`". Once the context has been created, you need to set the parent classloader. This is a very important step because the Web application may need access to the servlets in the container's jars. In this example, you need access to the `DefaultServlet` and `JSPServlet`, so setting the parent classloader gives the Web application access to these classes.

As stated before, this example does not use a default `web.xml` file. Therefore, it is important to set up the `DefaultServlet` and `JSPServlet` objects. This is done with a `Wrapper` object that is created from the `Context`. After you create a `Wrapper`, you set the servlet's initialization parameters, the servlet's class, and the order of loading it on startup. You then add the `Wrapper` to the `Context` and set up a servlet mapping. If you look at a default `web.xml` file, you will notice the code on lines 70–91 correlates to the `web.xml`'s servlet configuration. Once each servlet/`Wrapper` has been configured, it is added to the `Context` and the mapping for that servlet is set.

```
70              // Create a default servlet
71              Wrapper servlet = context.createWrapper();
72              servlet.setName("default");
73              servlet.setServletClass(
74                      "org.apache.catalina.servlets.DefaultServlet");
75              servlet.setLoadOnStartup(1);
76              servlet.addInitParameter("debug", "0");
77              servlet.addInitParameter("listings", "false");
78              context.addChild(servlet);
79              context.addServletMapping("/", "default");
80
81              // Create a handler for jsps
82              Wrapper jspServlet = context.createWrapper();
83              jspServlet.setName("jsp");
84              jspServlet.setServletClass(
85                      "org.apache.jasper.servlet.JspServlet");
86              jspServlet.addInitParameter("fork", "false");
87              jspServlet.addInitParameter("xpoweredBy", "false");
88              jspServlet.setLoadOnStartup(2);
89              context.addChild(jspServlet);
90              context.addServletMapping("*.jsp", "jsp");
91              context.addServletMapping("*.jspx", "jsp");
```

Because the default `web.xml` file also contains welcome files and MIME mappings, lines 93–100 show how to programmatically set these items as well. For the sake of brevity of this example, only the most common types were coded on lines 100–107. A more careful perusal of the default `web.xml` file will show many more MIME types. You can add or remove MIME types at your discretion.

```
93              // Set seme default welcome files
94              context.addWelcomeFile("index.html");
95              context.addWelcomeFile("index.htm");
96              context.addWelcomeFile("index.jsp");
97              context.setSessionTimeout(30);
98
99              // Add some mime mappings
100             context.addMimeMapping("html", "text/html");
101             context.addMimeMapping("htm", "text/html");
102             context.addMimeMapping("gif", "image/gif");
103             context.addMimeMapping("jpg", "image/jpeg");
104             context.addMimeMapping("png", "image/png");
105             context.addMimeMapping("js", "text/javascript");
106             context.addMimeMapping("css", "text/css");
107             context.addMimeMapping("pdf", "application/pdf");
```

Keep in mind that lines 70–107 are necessary only if your embedded application contains JSPs or you need the default servlet to handle error reporting (which generally is a good idea). If you are not using a default `web.xml` file, we recommend that you include the code on lines 70–107, but it is optional.

Once you have a context and have added it to the `Host`, you then need to create a `Connector` and add it to the `Embedded` object. Lines 55–61 have a `Connector` created with the `Embedded.createConnector()` method. It takes three parameters, and this particular method uses a `String` representing what address it listens on, the port to listen on, and whether it is a secure (HTTPS) type connector. It is very important to be careful about the address parameter. Many people make the mistake of setting this to "localhost" or

"127.0.0.1," which works, but then it allows connections from that machine only. This means your Web application may not communicate with anyone on another IP address. If you wish to allow access from any IP address, pass a `null`, or "0.0.0.0," which tells the `Connector` to listen on all IP addresses.

```
 55             // Create a connector that listens on all addresses
 56             // on port 5050
 57             Connector connector = embedded.createConnector(
 58                     (String)null, 5050, false);
 59
 60             // Wire up the connector
 61             embedded.addConnector(connector);
```

Last but not least, once the `embedded` object has been fully configured and wired with all of the objects, it can then be started, as shown on line 114.

```
114             embedded.start();
```

At this point, Tomcat begins processing Web requests.

The `docbase`, or directory where your application resides, will handle a full Java-based Web application, including servlets and JSPs (as long as the default objects have been set as shown on lines 70–107). You can set up a Web application with a WAR structure including the `WEB-INF` subdirectory. Also, the embedded Tomcat is multithreaded. When you start the embedded container, it returns immediately. On lines 136–138, the main application is put into a wait state to prevent it from exiting and allow the embedded container to process Web requests. In a more robust application, you may have the synchronized wait signaled from a shutdown command, or perhaps have the application sleep for a specificed amount of time, and awake to exit.

Running the MyWebServer Example

You can compile and run the MyWebServer example with Ant. You first must compile the application by typing the following command:

```
ant compile
```

The host's Web application directory is `myapps` and the application is underneath that in the `docbase` directory. That directory contains an `index.jsp`. To execute the application, you simply tell Ant to run it with:

```
ant run
```

After executing the application, you should see some output on your console:

```
Buildfile: build.xml
run:
     [java] Mar 13, 2007 4:46:03 PM org.apache.catalina.startup.Embedded start
     [java] INFO: Starting tomcat server
     [java] Mar 13, 2007 4:46:03 PM org.apache.catalina.core.StandardEngine start
     [java] INFO: Starting Servlet Engine: Apache Tomcat/6.0.10
     [java] Mar 13, 2007 4:46:03 PM
     org.apache.catalina.startup.ContextConfig defaultWebConfig
     [java] INFO: No default web.xml
```

```
        [java] Mar 13, 2007 4:46:03 PM org.apache.coyote.http11.Http11Protocol init
        [java] INFO: Initializing Coyote HTTP/1.1 on http-5050
        [java] Mar 13, 2007 4:46:03 PM org.apache.coyote.http11.Http11Protocol start
        [java] INFO: Starting Coyote HTTP/1.1 on http-5050
```

At this point the embedded application has started. To see if it works, open a browser and navigate to `http://localhost:5050`, and you should see a screen like the one shown in Figure 18-2.

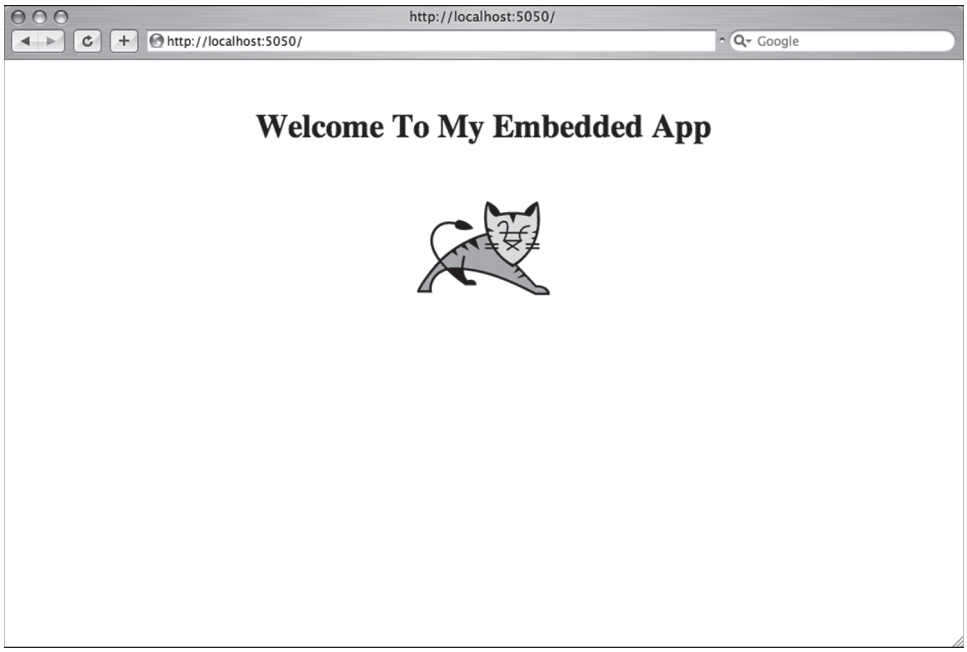

Figure 18-2: MyWebServer embedded application browser view

If you wish to compile the code in an IDE, all of the necessary libraries that come with the code are included with the code sample in the `lib` directory. These include `annotations-api.jar`, `el-api.jar`, `jasper.jar`, `servlet-api.jar`, `catalina.jar`, `jasperel.jar`, `jsp-api.jar`, `xercesImpl-2.8.1.jar`, `coyote.jar`, `jasper-jdt.jar`, and `juli.jar`. These will need to be added to your IDE's classpath to compile the code.

Summary

This chapter on embedded Tomcat included the following key points:

- Tomcat 6 formally supports embedded mode of operation. Using Tomcat in embedded mode, application developers can embed the functionality of a simple Web server, a servlet, and a JSP container within their own applications or products. Servlet and JSP support provides developers with a modular and reusable way of constructing their application logic. Tomcat 6's inherent

support for the HTTP protocol also makes it an attractive candidate for any application that requires a Web-based user interface.

- There are many uses for embedding Tomcat to allow you to have full control over the lifecycle and specify how you want Tomcat to serve your data. Uses include creating an application that serves secure Web content, providing a small Web server or HTTP services, or building an advanced implementation that makes your own fully working Web container, such as Apache Geronimo.
- Catalina, itself, is based on the embedded APIs and is a sophisticated implementation of the embedded API. Embedding Tomcat programmatically follows the way `service.xml` is put together. Catalina uses the Digester to dynamically use the embedded API to create a full-service Web container.
- Embedding Tomcat is a simple matter of creating an `Engine`, a `Host`, a `Context`, possibly one or more `Wrapper` objects for servlets, and linking them together to form relationships.

The capability to embed Tomcat in a custom application opens up a whole new world of possibilities for networked solutions developers, and brand-new opportunities for seasoned Tomcat developers.

19

Logging

Logging is critical for Web applications, as often there is no other way to debug issues on production machines. Also, adding log statements is the only viable option for analyzing the execution of a long-running Web application.

This chapter discusses logging, both internally by Tomcat as well as by Web applications themselves. This chapter covers the following:

- A short, tutorial-style introduction to both log4j and Java Logging.
- Logging concepts and best practices, such as rolling logs, maintaining information in log messages to be able to identify the source of the error, and much more.
- Solutions ("recipes") for common tasks that developers and system administrators need to do while developing effective logging solutions for their applications. Some of these include:
  - Logging to different destinations, such as a file, system console, or even the system logger (syslog, NT logger)
  - Being able to notify administrators via e-mail of specific log messages, such as serious errors

The chapter also introduces administrators to tools that they can use to manage the large amount of log data generated.

Changes from Tomcat 5

There are two big changes in logging since Tomcat 5:

- Tomcat 6 has done away with the `<Logger>` element defined in the `server.xml` that sent internal Tomcat messages to the `localhost_log`. Instead, you use the Java Logging property file `<CATALINA_HOME>/conf/logging.properties` to configure logging. See the section on JULI for more information on Java Logging configuration files.
- Tomcat 6 uses Java Logging instead of the Commons Logging API.

Tomcat 5 used the Commons Logging API (`jakarta.apache.org/commons/logging/`) instead of directly calling log4j or Java Logging. Commons Logging is an abstraction over these two logging implementations and allows developers to easily switch logging libraries. This API does a discovery when invoked, to determine which of the logging implementations to use. While this might seem to be a good idea — and a lot of other Apache projects still use this API — it has a number of drawbacks.

The main drawback is that Commons Logging ended up supporting the least common denominator of features between the logging APIs. There isn't a one-to-one mapping between log4j and Java Logging. In fact, some useful features of log4j, such as Nested Diagnostic Contexts, do not have an equivalent in Java Logging.

Second, this discovery process for a logging implementation added a lot of complexity to Commons Logging. This causes a lot of developer issues related to getting the right logging properties file loaded, class loading, and more. Finally, there was a runtime overhead, as calls to the logging API were wrapped by a Commons Logging call.

For these reasons and more, Tomcat 6 has moved to Java Logging. However, Java Logging itself is not without issues: the main problem being that the configuration for Java Logging is per JVM and not per class loader. Thus Java Logging cannot support different configurations for each Web application. To solve this issue, the Tomcat developers implemented their own "container-friendly" Java Logging implementation called JULI. This implementation allows you to have different logging property files for each Web application. In addition, it improves the features of the Java Logging property files. Tomcat 6 internally uses JULI, and so can Web applications deployed in Tomcat. As mentioned earlier, the logging properties file used by Tomcat is `<CATALINA_HOME>/conf/logging.properties`.

While JULI does solve the Java Logging issues, it introduces a non-standard Java Logging implementation. Thus, if you have to choose a logging implementation for your Web applications, log4j is still a good choice.

log4j

Java ships with its own Logging API (since JDK 1.4), and Tomcat 6 bundles its container-friendly implementation of this, called JULI. However, log4j, given its maturity and developer familiarity, is still a very widely used implementation.

log4j is an open-source Java framework for logging that you can download from `logging.apache.org/log4j`. This framework is currently a part of the Logging Services project of the Apache Software Foundation (ASF).

log4j can be configured easily to log messages at runtime without greatly affecting the performance of the application. It also provides control over the kind of information that is to be logged, without changing the application code. It makes use of a simple configuration file, and changing this file controls the logging behavior of log4j.

log4j Architecture

log4j uses a modular design that enables you to change the behavior of the application by modifying the log4j configuration file. It contains the following components, which are used to log information about the application state:

- Logger
- Appender
- Levels
- Filter
- Layouts

The following sections briefly describe each of these components.

Logger

Loggers are the main component of log4j, and they control the scope of logging. The application makes log requests on the Logger object. One or more loggers can be configured to log all messages as well as messages for a specific scope within the application — for example, the package or class.

Loggers live in a hierarchy, with `rootLogger` at the very top. Individual classes and packages can have loggers, too, with a logger for the package `foo.bar`, for example, being the parent of the logger of the package `foo.bar.baz`. This is useful as you will see later — missing settings in the log4j property files, such as for the debug level, get inherited from the parent logger.

An *Appender* must be assigned to a Logger (Appenders are discussed in the next section). The Appender controls the defined logging target — for example, a log file. Different kinds of Appenders can be assigned to the same Logger or to different Loggers. This means that different parts of an application can have different log destinations.

The following example shows what a Logger entry looks like in the log4j configuration file:

```
# Define a Logger named 'WroxLogger'. Assign the Level DEBUG to it. Assign an
# Appender 'WroxAppender' to this Logger
log4j.WroxLogger = DEBUG, WroxAppender
```

Appender

This log4j component manages the logging of information to an actual output location (for example, the console, a log file, and so on). The various kinds of log destinations are discussed later in this chapter. Every logging environment should have an Appender assigned. It is also possible to set different logging destinations for different parts of the application code. A layout is assigned to each Appender. This layout determines the actual formatting of the logged messages. Each Appender logs the message at the specified destination using the assigned layout.

log4j provides different kinds of Appenders. Every Appender is implemented by a class in the log4j API. For example, `ConsoleAppender` logs information to the console, `FileAppender` logs information to a file, and `SMTPAppender` logs information to an SMTP server.

In the following example, the information is logged to the console:

```
# Set the type for 'WroxAppender' as ConsoleAppender (writes to system console).
log4j.appender.WroxAppender = org.apache.log4j.ConsoleAppender
```

log4j is also capable of sending log messages to multiple destinations simultaneously (such as logging the same information to a file and the console).

Some of the other Appenders supported by log4j include:

- `JDBCAppender`: Writes logs to a database
- `JMSAppender`: Writes to a Java Message queue
- `NTEventLogAppender`: Writes to the NT Event log
- `SyslogAppender`: Writes to the UNIX/Linux syslog
- `L5Appender`: Writes to a Java Swing–based console
- `TelnetAppender`: Allows for a telnet-friendly way to monitor log messages
- `SocketAppender`: Writes to a network socket

This list of Appenders is quite exhaustive and should meet most requirements. If not, it is fairly easy for Java developers to implement a custom Appender.

Level

log4j uses various levels of logging. These are predefined priority values that can be used during logging. A log level refers to a particular priority of logging. log4j can be configured to log messages at the following levels:

- `TRACE`: This level is meant for debugging, similar to the `DEBUG` level, which follows. `TRACE` is intended to be used for an even finer degree of debug messages than `DEBUG`.
- `DEBUG`: This level is meant for logging debug messages. These messages are typically turned off in a production scenario.
- `INFO`: This level is used to track the progress of a running application by logging messages that give an indication of the application state or health.
- `WARN`: This level is used to log potentially harmful messages.
- `ERROR`: This level is used to log application error messages. By convention, these are meant to be logged when there are serious errors in an application, but the application can still recover from it and continue to run.
- `FATAL`: This level is meant to log messages that lead to an application crash or a state from which it cannot recover.

Following are the priority levels in ascending order:

```
TRACE < DEBUG < INFO < WARN < ERROR < FATAL
```

In addition to these levels, there are two special levels:

- `OFF`: This is the highest priority, where all logging is disabled.
- `ALL`: This is the lowest priority, where messages at all levels are logged.

Filter

Filters, as the name suggests, can be used to filter specific log messages. For instance, they can be used to allow or deny log messages that match a string pattern, or lie in specific range of priority levels. Filters can also be chained together.

log4j supports the filters listed here:

- `LevelMatchFilter`: Allow or deny log messages for a specific log level.
- `LevelRangeFilter`: Allow or deny log messages for a log level range.
- `StringMatchFilter`: Allow or deny log messages that match a string pattern.
- `DenyAllFilter`: Stop all log messages. This is typically used at the end of a filter chain to stop all messages that match a set of filters.

log4j supports configuration files that are both in a property file format as well as XML based, and these are covered in more detail later in the chapter. However, filters currently can only be configured using the XML format configuration files.

Layout

Layouts control the formatting of information. log4j can be configured to arrange the logged information in a concise and comprehensive manner. This way, the information that is logged is easy to read, making it usable and easier to interpret. log4j provides various classes that manage this, and these are discussed later in this chapter. The `SimpleLayout` layout logs messages in the simplest possible form.

Every layout component is associated with an Appender. Once the Appender sends the information to be logged to the specified destination, the layout takes care of formatting these log messages in the required format. The layout provides details such as date and time, name of the Java class, thread status, level, and so on.

Following is a sample entry of layout from the log4j configuration file:

```
# A log4j Appender 'WroxAppender' is defined as shown in the Appenders section.
# SimpleLayout is assigned to WroxAppender.
log4j.appender.WroxAppender.layout = org.apache.log4j.SimpleLayout
```

log4j Installation and Configuration

This section describes how to set up log4j for the actual development environment. First, download a stable release of log4j from `logging.apache.org/log4j`. All examples discussed in this chapter are tested with log4j version 1.2.14 with Tomcat 6 and Java SE.

Extract the downloaded file into a convenient directory on your local machine. This directory is denoted by `<Log4j_HOME>` in the remainder of this discussion.

It is assumed that the environment variables `PATH` and `CLASSPATH` for Java are properly set. The classes of log4j can be made available to the application by simply including the `log4j-x.y.z.jar` file in the `CLASSPATH`. This file is located under `<Log4j_HOME>\dist\lib`. Here, the `x.y.z` stands for the version number of log4j, as in `log4j-1.2.14.jar`.

To use log4j from your Web application, copy the log4j JAR file to the `<CATALINA_HOME>\lib` directory to make it available to all Web applications, or under your Web application's `WEB-INF\lib` if you want it for a specific Web application only.

log4j Configuration

log4j can be configured in the following ways:

- Using a simple properties file
- Programmatically
- Using an XML configuration file

Using a Simple Properties File

The simplest way to configure a Logger is to use a Java properties file typically referred to as a *Log Configuration File (LCF)*. It specifies all the configuration details. The entries are simple name-value pairs, and the pound symbol (#) is used to comment out a line.

The default name for the configuration file is `log4j.properties`. log4j will look for this configuration file in the `CLASSPATH`. You can specify the name and location of the property file, either via the `log4j.configuration` system property, or programmatically using the `PropertyConfigurator.configure()` method.

The following discussion shows how to build a sample log configuration file (`WroxSampleLog.properties`) and specify the path programmatically. You should first set up a Logger named `WroxSimpleLogger`. Assign an Appender named `wroxSimpleAppender` to this Logger. In addition, the logging level should be `DEBUG`:

```
log4j.logger.WroxSimpleLogger= DEBUG, wroxSimpleAppender
```

Once the Logger is in place, configure the Appender. Use the `FileAppender` to log all messages to a file. It is also needed to provide the name of the log file in this configuration:

```
log4j.appender.wroxSimpleAppender=org.apache.log4j.FileAppender
log4j.appender.wroxSimpleAppender.File= wroxSimpleLog.log
```

Finally, set the desired layout for the logging. This example uses `SimpleLayout`:

```
log4j.appender.wroxSimpleAppender.layout=org.apache.log4j.SimpleLayout
```

Following is the complete sample log configuration file:

```
log4j.logger.WroxSimpleLogger = DEBUG, wroxSimpleAppender
log4j.appender.wroxSimpleAppender = org.apache.log4j.FileAppender
log4j.appender.wroxSimpleAppender.File = c:\wroxSimpleLog.log
log4j.appender.wroxSimpleAppender.layout = org.apache.log4j.SimpleLayout
```

The configuration file must be specified to the `PropertyConfigurator`, which enables it to read the configuration. In the application code, a simple statement such as the following does this:

```
PropertyConfigurator.configure("WroxSampleLog.properties");
```

Once the configuration is loaded, each class that requires logging capability within the application must have a reference to the Logger object. Because the Logger uses a package naming convention, it is common among developers to use the class name as the Logger, although this is not mandatory. In any case, a package style of Logger naming serves well. To use log4j in the application, the property file is loaded by using a `PropertyConfigurator`. This is shown in the following line — substitute the actual path on your system in place of `path_to_file`.

```
import org.apache.log4j.PropertyConfigurator;
...
PropertyConfigurator.configure("path_to_file/WroxSampleLog.properties");
```

Another interesting variant of this is the `PropertyConfigurator.configureAndWatch()` method. This allows you to have the changed log4j property file reloaded without needing to restart the application. The reload happens by checking for changes to the file after every minute.

```
PropertyConfigurator.configureAndWatch("path_to_file/WroxSampleLog.properties");
```

This default delay can be changed, too, by passing your own desired delay time (in milliseconds):

```
// Check for changes in the log4j configuration file every 5 minutes
PropertyConfigurator.configureAndWatch("path_to_file/WroxSampleLog.properties",
                                       5 * 60000);
```

Using log4j Programmatically

log4j can also be configured programmatically. In the application code, a new `Logger` object is created, and is configured in much the same way as the configuration file. The main steps for using log4j programmatically are as follows:

1. In your application code, create a Logger object. Typically, the package hierarchy — including the class name — is used:

```
Logger WroxTestLogger = Logger.getLogger("wroxLogging.WroxLoggingTest");
```

2. The next step is to set the logging level in the Logger object:

```
WroxTestLogger.setLevel(Level.DEBUG);
```

3. Create an instance of the layout to be used:

```
SimpleLayout WroxSimpleLayout = new SimpleLayout();
```

4. Create an instance of the Appender to be used. While creating the Appender, the layout is set to the `WroxSimpleLayout` created earlier and, because this is a FileAppender (i.e., one that writes to a log file), also sets the log file.

```
FileAppender WroxFileAppender =
             new FileAppender(WroxSimpleLayout,"WroxSampleLog.log");
```

5. Add this Appender object to the Logger object:

```
WroxTestLogger.addAppender(WroxFileAppender);
```

The following completed sample Java class shows a Logger being created and configured to log messages:

```
package wroxLogging;
import org.apache.log4j.Logger;
import org.apache.log4j.Level;
import org.apache.log4j.SimpleLayout;
import org.apache.log4j.FileAppender;
import java.io.IOException;
public class WroxLoggingTest {
public static void main(String[] args) {
    //Create an instance of Logger
    Logger WroxTestLogger = Logger.getLogger("wroxLogging.WroxLoggingTest");
    //Set the Logging Level
    WroxTestLogger.setLevel(Level.DEBUG);
    try {
      SimpleLayout WroxSimpleLayout = new SimpleLayout();
      //Assign a SimpleLayout to the FileAppender
     FileAppender WroxFileAppender=
             new FileAppender(WroxSimpleLayout,"WroxSampleLog.log");
      //Finally Assign the fileAppender to the Logger
      WroxTestLogger.addAppender(WroxFileAppender);
      //Now try to Log few messages at different 'Levels'
      WroxTestLogger.debug("Sample Message : DEBUG";);
      WroxTestLogger.info("Sample Message : INFO");
      WroxTestLogger.warn("Sample Message : WARN");
      WroxTestLogger.error("Sample Message : ERROR");
      WroxTestLogger.fatal("Sample Message : FATAL");
    } catch(IOException e) {
      WroxTestLogger.warn("An IOException was thrown", e);
    }
  }
}
```

In practice, a programmatic configuration for log4j is rarely used; using a property- or XML file–driven configuration allows you to change log4j behavior without changing code.

Using an XML Configuration File

The log4j configuration file can also be in XML format in much the same way as the properties file. The only difference is that the log4j class loading the XML configuration is `DOMConfigurator`, instead of `PropertyConfigurator`.

The file begins with a standard XML declaration and the DTD declaration. This ensures that the XML configuration file conforms to the syntax of log4j:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
```

The root element in the XML file is `<log4j:configuration>`. The `log4j` portion is the namespace used for the configuration file. The root element `<log4j:configuration>` wraps the entire configuration. It has two main subelements, `appender` and `logger`.

```
<log4j:configuration>
```

The main attributes of the `appender` element are `name` and `class`. It also takes parameters such as `Layout` as subelements. The following is a sample entry:

```
  <appender name="wroxFileAppender" class="org.apache.log4j.FileAppender">
    <param name="File" value="log4j.log"/>
    <layout class="org.apache.log4j.SimpleLayout"/>
  </appender>
```

The next important element is `<logger>`. It also has attributes and a few subelements. The main attribute is `name`. The parameter it takes is the associated `appender` element. A typical entry looks like the following:

```
  <logger name="wroxLogging.WroxSimpleLogger">
    <level value="debug"/>
    <appender-ref ref="wroxFileAppender" />
  </logger>
```

Here is the complete sample configuration file `WroxSampleConfig.xml` for log4j in XML format:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration>
  <appender name="wroxFileAppender" class="org.apache.log4j.FileAppender">
    <param name="File" value="log4j.log"/>
    <layout class="org.apache.log4j.SimpleLayout"/>
  </appender>
  <logger name="wroxLogging.WroxSimpleLogger">
    <level value="debug"/>
    <appender-ref ref="wroxFileAppender" />
  </logger>
</log4j:configuration>
```

Once the configuration is provided in the XML file, the `DOMConfigurator` can be used to load these settings. The `DOMConfigurator` looks for the XML configuration file in the same directory as that of the application. `DOMConfigurator` can also use a specific system path or URL:

```
public static void main(String[] args) {
  // Load the configuration through the DOMConfigurator and provide in the XML file
  DOMConfigurator.configure("WroxSampleConfig.xml");
  Logger WroxLogger = Logger.getLogger("wroxLogging.WroxSimpleLogger");
  // Now try to log few messages at different Levels
  WroxLogger.debug("Sample Message : DEBUG");
  WroxLogger.info("Sample Message : INFO");
  WroxLogger.warn("Sample Message : WARN");
  WroxLogger.error("Sample Message : ERROR");
  WroxLogger.fatal("Sample Message : FATAL");
}
```

A Tutorial Introduction to log4j

This section presents a simple tutorial that shows you how to get started with log4j. Other advanced options and settings are discussed in the section "More log4j Recipes.".

The tutorial shows you how to log messages to the console as well as a log file from your application. This involves three steps:

1. Ensure that the log4j JAR file is in your `CLASSPATH`.
2. Create a `log4j.properties` property file and ensure that it is in your `CLASSPATH`, too.
3. Add log statements to your application.

As shown in the section "log4j Installation and Configuration," download log4j and add the JAR file to your `CLASSPATH`.

Next, create a property file for log4j. The default name for this file is `log4j.properties`, and if you follow this convention, log4j will be able to pick this file up from your `CLASSPATH` without your having to specify which configuration file to read.

The sample log configuration file is shown next. As you can see, the log level is set to `WARN`, and so only messages with a level equal to or more — i.e., `WARN`, `ERROR`, and `FATAL` — will get logged. As mentioned earlier in the chapter, rootLogger is a special logger that always exists, and is at the top of the logger hierarchy. The root logger, in the following sample configuration, sends the log messages to an Appender called `myConsoleAppender`, which sends the message to the console.

```
# Define a Logger that sends log messages to the console and a log file
log4j.rootLogger = WARN, myConsoleAppender
# Define the console Appender
log4j.appender.myConsoleAppender = org.apache.log4j.ConsoleAppender
# Assign a Layout to the Appenders
log4j.appender.myConsoleAppender.layout = org.apache.log4j.SimpleLayout
```

Finally, add the following code to the log from your application:

```
package myPackage;
import org.apache.log4j.Logger;
public class LoggingExample {
    /* Create a logger for this class */
    private static Logger log = Logger.getLogger(LoggingExample.class);
public static void main(String[] args) {
        /* Log messages with varying severity levels */
        log.trace("This is a trace message");
        log.debug("This is a debug message");
        log.info("This is an info message");
        log.warn("This is a warning message");
        log.error("This is an error message");
        log.fatal("This is a fatal error. All hope is lost!");
    }
}
```

In the application code, the `Logger.getLogger()` method call gets the logger object for the class. The `getLogger()` method takes either a name for the logger, or a class object that it converts to a name. You can pass it any arbitrary name, but the convention is to use the class name. The Logger object is then used to invoke log methods to write out trace, debug, information, warning, error, or fatal messages — for example, `log.error("...")`.

Does the Logger object have to be `static`? This is the common usage pattern for creating a Logger object. Programmers do this because, in most cases, the Logger does not need to distinguish between instances of the class (here, the `LoggingExample` class), so it helps save on memory. If you don't declare the Logger as static, each instance of `LoggingExample` may have its own Logger. However, in current log4j implementations, the `getLogger()` method returns the same instance of the Logger for the same class name, so this is not really required.

Compile this class, and run it. As mentioned before, ensure that the log4j jar file, the log4j property file, and the application code are in the `CLASSPATH`.

All the logged messages appear on the console as shown:

```
WARN - This is a warning message
ERROR - This is an error message
FATAL - This is a fatal error. All hope is lost!
```

Now experiment with the logging levels: Move the level up from `WARN` to `ERROR` or lower it to `TRACE`, and see the amount of logging messages output to the console and the log file change.

If this Java class is inside a Web application, then the same principle applies: Copy the `log4j.jar` file to the `<CATALINA_HOME>/lib` or `WEB-INF/lib` directory of the Web application and place the Web application–specific `log4j.properties` under `WEB-INF/classes`.

More log4j Recipes

The log configuration file stores all the log4j options that control what information should be logged, and how it should be logged. The following sections provide details about various logging options.

Logging from a Web Application

As described earlier, logging from a Web application works the same way as logging from a standalone Java class; ensure that the `log4j.properties` file and the log4j JAR file are in your `CLASSPATH`, and add log statements to your source code — be that servlet code, JSPs, or any Java classes called by the servlets or JSPs. This requires copying the `log4j-x.y.z.jar` file to the `WEB-INF/lib` directory of the Web application and placing the log4j.properties under `WEB-INF/classes`.

You can also have log4j configuration initialized programmatically via an initialization servlet. This way, the properties are read at the time the application is started. If you are doing this, then the `<load-on-startup>` tag in the `web.xml` file should be set to `1`. This ensures that the initialization servlet is invoked at the time of Web application startup. Once the log4j is configured, any JSP, servlet, or Java class within that Web application can use log4j features.

Here is the complete source code of the initialization servlet:

```
package wroxLogging;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import org.apache.log4j.PropertyConfigurator;
public class WroxLogServlet extends HttpServlet {
  public void init()
    throws ServletException {
    // Get Fully Qualified Path to Properties File
    String config = getServletContext().getRealPath("/") +
                    getInitParameter("setup");
    System.out.println("LoggingServlet Initialized using file :" + config);
    // Initialize Properties for All Servlets
    PropertyConfigurator.configure(config);
  }
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    /* Do nothing */
    PrintWriter out = response.getWriter();
    response.setContentType("text/html");
    out.println("<html>");
    out.println("<head>");
    out.println("<title>LoggingServlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<p>Called LoggingServlet </p>");
    out.println("</body>");
    out.println("</html>");
    out.close();
  }
  public void doPost(HttpServletRequest request, HttpServletResponse response)
     throws IOException, ServletException {
       /* Do nothing */
       doGet(request, response);
  }

  public void destroy() {
  }
}
```

Following are the `web.xml` settings:

```
<servlet>
    <servlet-name>WroxLogServlet</servlet-name>
    <servlet-class>wroxLogging.WroxLogServlet</servlet-class>
    <init-param>
        <param-name>setup</param-name>
        <param-value>WEB-INF/WroxLogging.properties</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
 </servlet>
```

Logging to the Console

The `ConsoleAppender` was introduced earlier in the chapter in the log4j tutorial. This Appender is used to log messages to the console. As the sample log4j configuration file shows, there are additional tunable parameters for this Appender, such as the logging behavior (flush immediately to console, instead of the default buffering), as well as the log level (`log4j.appender.<Appender name>.Threshold`). This threshold setting overrides all parent settings — in this case the settings for the root logger — and limits the log level of messages that are output. Make the following changes to the log4j property file used in the tutorial, and run the LoggingExample class as before to see the effect.

```
# Define a Logger that sends log messages to the console
log4j.rootLogger = ALL, myConsoleAppender
# Define the console Appender
log4j.appender.myConsoleAppender = org.apache.log4j.ConsoleAppender
log4j.appender.myConsoleAppender.Threshold = WARN
log4j.appender.myConsoleAppender.ImmediateFlush=true
# Specify which output stream to be used, standard or error
log4j.appender.myConsoleAppender.Target=System.err
# Assign a Layout to the Appender
log4j.appender.myConsoleAppender.layout = org.apache.log4j.SimpleLayout
```

Logging to a File

The `FileAppender` lets you send messages to a log file. Some useful attributes of this Appender are introduced in the next example. These include options to set the log filename (`log4j.appender.<appender name>.File`) and the logging level handled by this Appender (`log4j.appender.<appender name>.Threshold`), to flush immediately or not (`log4j.appender.<appender name>.ImmediateFlush`), and to append to the log file or overwrite it each time (`log4j.appender.<appender name>.Append`).

The following sample shows a `FileAppender` with these options:

```
# Define a Logger that sends log messages to the console
log4j.rootLogger = ALL, myFileAppender
# Define the log file Appender
log4j.appender.myFileAppender = org.apache.log4j.FileAppender
log4j.appender.myFileAppender.File = C:/tmp/messages.log
log4j.appender.myFileAppender.Threshold = WARN
log4j.appender.myFileAppender.ImmediateFlush=true
# Append to existing contents or overwrite
log4j.appender.myFileAppender.Append=false
# Assign a Layout to the appender
log4j.appender.myFileAppender.layout = org.apache.log4j.SimpleLayout
```

Logging to Multiple Destinations

The tutorial section shows an example of logging messages to the console. To log to multiple destinations, simply add another Appender as shown. In this example, messages are logged at two separate targets: the system console and a log file.

The following is the log4j configuration file:

```
# Define a Logger that sends log messages to the console and a log file
log4j.rootLogger = WARN, myFileAppender, myConsoleAppender
# Define the log file Appender
log4j.appender.myFileAppender = org.apache.log4j.FileAppender
log4j.appender.myFileAppender.file = C:/tmp/messages.log
# Define the console Appender
log4j.appender.myConsoleAppender = org.apache.log4j.ConsoleAppender
# Assign a Layout to both Appenders
log4j.appender.myConsoleAppender.layout = org.apache.log4j.SimpleLayout
log4j.appender.myFileAppender.layout = org.apache.log4j.SimpleLayout
```

Rolling Log Files by Size

A common problem with log files is that a lot of log data gets generated over a period of time. The `RollingFileAppender` provides various ways to manage this problem, by allowing you to limit the size of the log file (`log4j.appender.<appender name>.MaxFileSize`), as well as "rolling" the older log messages to backup files. You can also control how many versions of these backup log files will be created using the `log4j.appender.<appender name>.MaxBackupIndex` parameter.

Here is a sample configuration for `RollingFileAppender`:

```
# Define a Logger that sends log messages to rolling log file
log4j.rootLogger = ALL, myRollingFileAppender
# Define the log file Appender
log4j.appender.myRollingFileAppender = org.apache.log4j.RollingFileAppender
log4j.appender.myRollingFileAppender.File = C:/tmp/messages.log
log4j.appender.myRollingFileAppender.Threshold = WARN
log4j.appender.myRollingFileAppender.MaxFileSize=100KB
log4j.appender.myRollingFileAppender.MaxBackupIndex=4
# Assign a Layout to the appender
log4j.appender.myRollingFileAppender.layout = org.apache.log4j.SimpleLayout
```

Rolling Log File by Date

Often it is useful to be able to get logs for a particular time period — say a specific day, week, or month. This becomes difficult if the messages all go to the same log file, as scripts need to be written to get log messages by a date/time stamp, assuming that this timestamp was output as a part of the log messages.

A cleaner solution is to use a `DailyRollingFileAppender`. Like the `RollingFileAppender` appender, this allows for rolling of log files; however, this is done based not on the size, but on a timestamp.

```
# Define a Logger that sends log messages to rolling log file
log4j.rootLogger = ALL, myRollingFileAppender
# Define the log file Appender
log4j.appender.myRollingFileAppender = org.apache.log4j.DailyRollingFileAppender
log4j.appender.myRollingFileAppender.File = C:/tmp/messages.log
log4j.appender.myRollingFileAppender.Threshold = ALL
log4j.appender.myRollingFileAppender.Append = true
log4j.appender.myRollingFileAppender.DatePattern = '.'yyyy-MM-dd
log4j.appender.myRollingFileAppender.layout = org.apache.log4j.SimpleLayout
```

The `DatePattern` option controls the rolling of the file — in this case, files are rolled over daily at midnight. The log filenames have the prefix `message.log` because that was specified in the Appender's File attribute, and the suffix in the yyyy-MM-dd format (for example, `message.log.2007-03-11`). Some other possible date patterns are listed in the table that follows.

Format	Effect
`'.'yyyy-MM`	Roll at the beginning of each month.
`'.'yyyy-ww`	Roll at the beginning of each week.
`'.'yyyy-MM-dd`	Roll once a day, at midnight.
`'.'yyyy-MM-dd-a`	Roll twice a day, at midnight and at noon.
`'.'yyyy-MM-dd-HH`	Roll every hour.

Other permutations of this are possible, too: Any format that conforms to the Java `SimpleDateFormat` conventions is allowed; however, the colon (:) character should not be used.

Separating Log Messages by Level

You could have a situation in which a log message of a specific severity level — say ERROR and FATAL — that needs attention by a system administrator should be logged in a separate log file from all other log messages.

The following log4j configuration file uses the Threshold parameter of the Appender to specify that only messages with a level ERROR or higher (i.e., ERROR and FATAL) get sent to the sysadmin's log file. The other Appender does not have a threshold specified, so it logs everything.

```
# Define a Logger that sends log messages to the console and a log file
log4j.rootLogger = ALL, myFileAppenderForSysadmin, myFileAppenderForDevelopers
# Define the log file Appender for the sysadmin
log4j.appender.myFileAppenderForSysadmin = org.apache.log4j.FileAppender
log4j.appender.myFileAppenderForSysadmin.File = C:/tmp/messages-s.log
log4j.appender.myFileAppenderForSysadmin.Threshold = ERROR
# Define the log file Appender for the developers
log4j.appender.myFileAppenderForDevelopers = org.apache.log4j.FileAppender
log4j.appender.myFileAppenderForDevelopers.File = C:/tmp/messages-d.log
# Assign a Layout to both Appenders
log4j.appender.myFileAppenderForSysadmin.layout = org.apache.log4j.SimpleLayout
log4j.appender.myFileAppenderForDevelopers.layout = org.apache.log4j.SimpleLayout
```

In most cases, this is enough. However, log4j allows an even more fine-grained mechanism to control log levels. This is done using Filters.

As discussed earlier, Filters enable you to allow or block log messages based on a specific level, level range, or even string patterns in the log message.

The following log4j configuration file shows how to achieve the same effect using the XML configuration file format. Again, as noted earlier, currently filters can be configured only by using the XML format.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration>
    <appender name="myFileAppenderForSysadmin" class="org.apache.log4j.
FileAppender">
        <param name="File" value="C:/tmp/messages-s.log"/>
        <layout class="org.apache.log4j.SimpleLayout"/>
        <filter class="org.apache.log4j.varia.LevelRangeFilter">
            <param name="LevelMin" value="ERROR"/>
            <param name="LevelMax" value="FATAL"/>
        </filter>
    </appender>
    <appender name="myFileAppenderForDevelopers" class="org.apache.log4j.
FileAppender">
        <param name="File" value="C:/tmp/messages-d.log"/>
        <layout class="org.apache.log4j.SimpleLayout"/>
        <filter class="org.apache.log4j.varia.LevelRangeFilter">
            <param name="LevelMin" value="DEBUG"/>
            <param name="LevelMax" value="FATAL"/>
        </filter>
    </appender>
    <root>
        <level value="all" />
        <appender-ref ref="myFileAppenderForSysadmin"/>
        <appender-ref ref="myFileAppenderForDevelopers"/>
    </root>
</log4j:configuration>
```

Enabling Logging for Specific Packages or Classes in the Application

You can turn on or off logging for specific packages or classes with your application. This can be done by defining a logger for a specific package (here the `myPackage` package), as shown in the following log4j properties file:

```
log4j.logger.myPackage = WARN, myConsoleAppender
log4j.appender.myConsoleAppender = org.apache.log4j.ConsoleAppender
log4j.appender.myConsoleAppender.layout = org.apache.log4j.SimpleLayout
```

You can even go further, and define a Logger for a specific class (here `myPackage.LoggingExample`), as shown:

```
log4j.logger.myPackage.LoggingExample = WARN, myConsoleAppender
log4j.appender.myConsoleAppender = org.apache.log4j.ConsoleAppender
log4j.appender.myConsoleAppender.layout = org.apache.log4j.SimpleLayout
```

Custom Formatting of Log Messages

log4j enables the customized formatting of log messages using the `PatternLayout`. This provides an option to log messages with detailed information. The following code snippet shows how to use the `PatternLayout` in the configuration file:

```
# Use the 'PatternLayout' as the layout
log4j.appender.myFileAppender.layout = org.apache.log4j.PatternLayout
```

Specifying the format of log messages is done via *format modifiers*. Following are some of the most commonly used format modifiers:

- `%m`: The actual log message
- `%n`: New line
- `%c`: Logger name
- `%t`: Current thread
- `%p`: Log level for the message (`DEBUG/INFO/WARN/ERROR/FATAL`)
- `%r`: Time (in milliseconds) since the code was executed

The default format modifier is `%m%n`. If the data item requires fewer characters, it is padded with space(s) on either the left or the right until the minimum width is reached. If the data item is larger than the minimum field width, the field is expanded to accommodate the data. In addition, a period followed by a positive integer to specify the maximum field width is also valid. The following lines show some samples of the commonly used combinations with `PatternLayout`:

```
log4j.appender.myFileAppender.layout.ConversionPattern = %-5p: %m%n
log4j.appender.anotherFileAppender.layout.ConversionPattern=[%d{yyyy-mm-dd hh:
mm}]%-5p[%t]%x(%F:%L) – %m%n
```

The first conversion pattern prints messages in the format:

```
WARN : This is a warning message
ERROR: This is an error message
FATAL: This is a fatal error. All hope is lost!
```

The second conversion pattern shows a lot more information; including timestamp, error level, current thread, and source file information:

```
[2007-19-11 11:19]WARN [main](LoggingExample.java:14) - This is a warning message
[2007-19-11 11:19]ERROR[main](LoggingExample.java:15) - This is an error message
[2007-19-11 11:19]FATAL[main](LoggingExample.java:16) - This is a fatal error.
All hope is lost!
```

Format modifiers can be used between the percentage symbol and the conversion character to change the minimum field width, the maximum field width, and text justification within a field. Use the minus sign (–) to left-justify within a field. By default, it is right-justified (pad on left). Use a positive integer to specify the minimum field width.

Logging Messages as HTML

The previous section explains how to configure log4j to log information in a variety of formats. This information can be logged, not just in a text format, but even as HTML. This is useful when the logs are generated for viewing via a Web browser. This format improves the overall presentation and readability of log files.

The following is the log4j configuration file:

```
# Define a Logger that sends log messages to the console
log4j.rootLogger = ALL, myFileAppender
# Define the log file Appender
log4j.appender.myFileAppender = org.apache.log4j.FileAppender
log4j.appender.myFileAppender.File = C:/tmp/messages.html
log4j.appender.myFileAppender.Threshold = WARN
log4j.appender.myFileAppender.ImmediateFlush=true
log4j.appender.myFileAppender.Append=false
# Assign a Layout to the appender
log4j.appender.myFileAppender.layout = org.apache.log4j.HTMLLayout
log4j.appender.myFileAppender.layout.LocationInfo = true
log4j.appender.myFileAppender.layout.Title = myApplication log messages
```

As you can see, the output file is now an HTML file, and uses an `HTMLLayout`. A couple of new attributes were used in the layout: `LocationInfo`, which prints line number information about the source of the log message; and `Title`, which specifies the title to the HTML page.

When the application is run with this log4j property file, the output log data can be viewed properly formatted for a browser (see Figure 19-1).

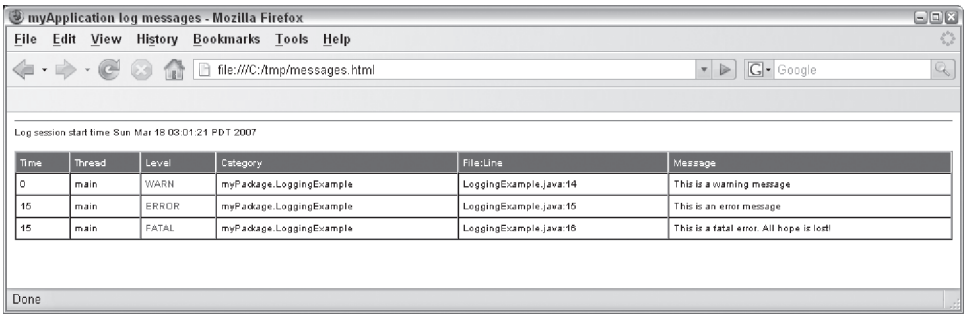
myApplication log messages - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/tmp/messages.html
Google
Log session start time Sun Mar 18 03:01:21 PDT 2007
Time Thread Level Category File:Line Message
0 main WARN myPackage.LoggingExample LoggingExample.java:14 This is a warning message
15 main ERROR myPackage.LoggingExample LoggingExample.java:15 This is an error message
15 main FATAL myPackage.LoggingExample LoggingExample.java:16 This is a fatal error. All hope is lost!
Done

Figure 19-1: Sample logging output in HTML

E-mail Log Messages

A very powerful feature of log4j is its capability to send log messages as e-mail via the SMTPAppender. This makes it possible for administrators to get critical messages as alerts. For example, `FATAL` level messages can be sent to an SMTP server, which can get the immediate attention of the server administrator. Although it might look complicated, the log4j implementation hides the complexities from the developer. This makes it easy to deploy this solution, as demonstrated in the following example.

To be able to use the SMTPAppender, you need JavaMail (`java.sun.com/products/javamail/`) and Java Activation Framework (`java.sun.com/products/javabeans/jaf/downloads/`) JAR files — `mail.jar` and `activation.jar` respectively — in your `CLASSPATH`. Web applications can be

copied to the application's `WEB-INF/lib` directory. These JAR files are needed in order for the JavaMail functionality to work.

The sample log4j configuration file for this example is shown next. The `To` attribute specifies the recipient of the log messages, the `SMTPHost` specifies the SMTP mail server to be used to send the e-mail, and the `Subject` and `From` attributes specify the subject and the sending e-mail address. The `BufferSize` specifies the number of log events that are saved in the SMTPAppender's buffer. Note the use of the `Threshold` attribute; this enables only error messages with ERROR or FATAL level to be sent to the sys admin.

```
Define the Logger :
log4j.rootLogger = ALL, myMailAppender
# Define the Appender
log4j.appender.myMailAppender = org.apache.log4j.net.SMTPAppender
log4j.appender.myMailAppender.To = sysadmin@mydomain.dom
log4j.appender.myMailAppender.SMTPHost = smtp.mydomain.dom
log4j.appender.myMailAppender.Subject = Error from myApplication
log4j.appender.myMailAppender.From = tomcat@mydomain.dom
log4j.appender.myMailAppender.BufferSize = 10
log4j.appender.myMailAppender.Threshold = ERROR
# Assign a Layout to the Appender
log4j.appender.myMailAppender.layout = org.apache.log4j.PatternLayout
log4j.appender.myMailAppender.layout.ConversionPattern = %d %-5p [%t] %c{2} - %m%n
```

See the section "Separating Log Messages by Level" to learn more. These can be used for an even more fine-grained control on the messages sent to the system administrator.

Logging to the NT Event Log

Some administrators prefer to integrate the application log and system log for convenience and maintenance. log4j can use the native system log (such as the syslog on Linux/Unix and the Windows Application log on Windows).

> *Logging to the Linux/Unix syslog, although not shown here, can be done by using the SyslogAppender (`org.apache.log4j.net.SyslogAppender`) instead of the NTEventLogAppender in the log4j configuration.*

A communication channel with the native operating system is required for this. For Windows, the log4j distribution provides a Dynamic Link Library (DLL), `NTEventLogAppender.dll`, to enable logging to the Event log.

The first step is registering this DLL. The `NTEventLogAppender` DLL is located under the `<LOG4J_HOME>\dist\lib` directory.

Copy the `NTEventLogAppender.dll` file into the `system32` directory under the Windows installation directory. For Windows XP the `system32` folder is located at `C:\WINDOWS\system32`. After copying the DLL file, execute the following command to register the DLL (the command can be executed by selecting Run from the Start menu):

```
regsvr32 C:\WINDOWS\system32\NTEventLogAppender.dll
```

The DLL registration step is shown in Figure 19-2.

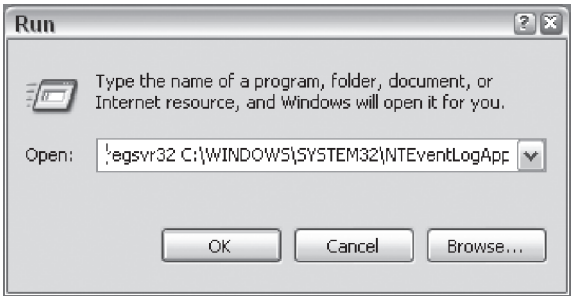

Figure 19-2: Registering the NTEventLogAppender DLL

When you register this DLL, you might get a "DllRegisterServer entry point was not found" error, as shown in Figure 19-3. This error can be ignored, and should go away when a fix for this is added to a later version of log4j.

RegSvr32
C:\WINDOWS\SYSTEM32\NTEventLogAppender.dll was loaded, but the DllRegisterServer entry point was not found.
This file can not be registered.
OK

Figure 19-3: Expected DllRegisterServer entry point error message

The following code shows the log configuration file:

```
# Define a Logger that sends log messages to the console
log4j.rootLogger = ALL, myNTEventLogAppender
# Define the console Appender
log4j.appender.myNTEventLogAppender = org.apache.log4j.nt.NTEventLogAppender
log4j.appender.myNTEventLogAppender.Threshold = WARN
# Assign a Layout to the appender
log4j.appender.myNTEventLogAppender.layout = org.apache.log4j.SimpleLayout
```

After registering the DLL, run the application code. The log messages will now be sent to the Windows native Application log. Once this is done, the entries generated in the Windows system Application log can be reviewed through the Event Viewer. You can use it by selecting Start➪Control Panel➪Administrative Tools➪Event Viewer. All the messages logged can be found under the Application log, as shown in Figure 19-4.

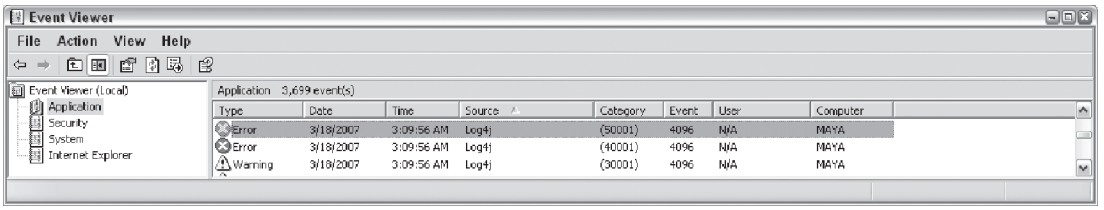
Event Viewer
File Action View Help
Event Viewer (Local)
Application
Security
System
Internet Explorer
Application 3,699 event(s)
Type Date Time Source Category Event User Computer
Error 3/18/2007 3:09:56 AM Log4j (50001) 4096 N/A MAYA
Error 3/18/2007 3:09:56 AM Log4j (40001) 4096 N/A MAYA
Warning 3/18/2007 3:09:56 AM Log4j (30001) 4096 N/A MAYA

Figure 19-4: The Windows Event Viewer

When you click any one entry from the Application log, all the details for that entry are displayed. This includes the application name and the actual log message, with a date and time stamp, as shown in Figure 19-5.

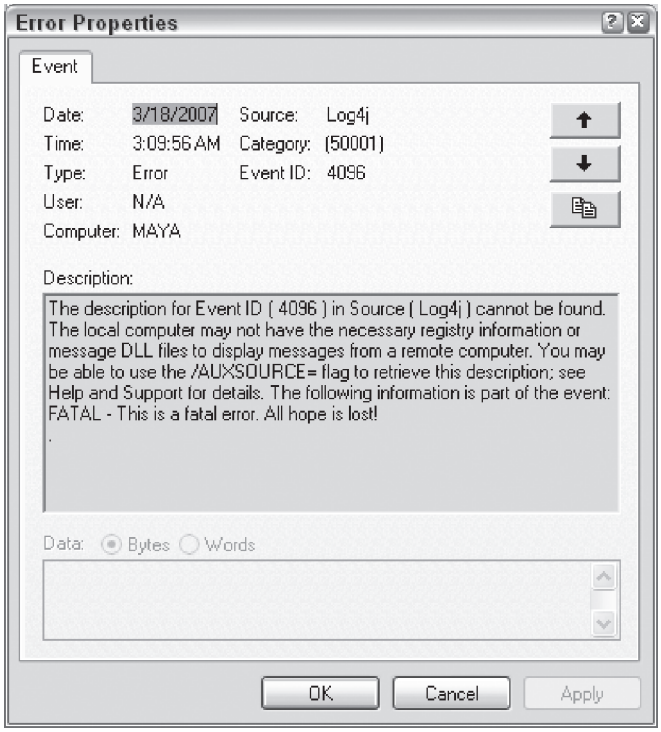
Error Properties
Event
Date: 3/18/2007
Source: Log4j
Time: 3:09:56 AM
Category: (50001)
Type: Error
Event ID: 4096
User: N/A
Computer: MAYA
Description:
The description for Event ID (4096) in Source (Log4j) cannot be found. The local computer may not have the necessary registry information or message DLL files to display messages from a remote computer. You may be able to use the /AUXSOURCE= flag to retrieve this description; see Help and Support for details. The following information is part of the event: FATAL - This is a fatal error. All hope is lost!
Data: Bytes Words
OK
Cancel
Apply

Figure 19-5: Log message details

Adding Additional Context Information Using Nested Diagnostic Context

To debug using a log file, it is important to be able to isolate log messages, such as all logs for a specific application, user session, and so on. This can get tricky when code outputting the log messages is shared between multiple applications. One approach to solving this problem is the *Nested Diagnostic Context* (NDC) supported by log4j.

An NDC is a mechanism to distinguish interleaved log messages. This is done by having each invocation context — for example, each user session — log a distinctive message stamp.

The log4j NDC implementation in log4j is based on a stack. To start using this, first push identifying information onto the NDC stack. This information typically can include information such as the client's IP address, identifying information from the cookie sent by the client, the session's unique ID (`HttpSession`'s `getId()` method), and more.

Once this is done, the `%x` pattern in your Appender's layout can print out this information to the log file. The following code snippet illustrates this usage:

```
import org.apache.log4j.NDC
...
public class SampleServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        ...
        String ndcContextString = request.getRemoteAddr()
                                  + ":" + request.getSession().getId();
        NDC.push (ndcContextString);
        ...
        log.error ("Some error message");
        ...
        NDC.pop ()
    }
    ...
}
```

The log4j properties file can now use the `%x` pattern to print this context information.

```
log4j.rootLogger = ALL, myConsoleAppender
log4j.appender.myConsoleAppender = org.apache.log4j.ConsoleAppender
log4j.appender.myConsoleAppender.layout = org.apache.log4j.PatternLayout
log4j.appender.myConsoleAppender.layout.ConversionPattern =
%-5p: [%d] %x %c{1} - %m%n
```

Beginning JavaServer Pages (Wrox, ISBN 978-0-7645-7485-6) demonstrates an example of a Tomcat filter that implements NDC, thus allowing you to add extra context information at will by changing configuration files.

View or Query Log Files

Any Web application running over a period of time generates tons of data. Without effective visual ways of viewing, querying, and tracing through this data, the utility of logging it is often lost.

Chainsaw (`logging.apache.org/log4j/docs/chainsaw.html`) is a companion project to log4j, and allows you to do this, and much more:

- Support for filtering log messages using a query language.
- The ability to specify different colors for log events based on their attributes, such as level, logger, and so on.
- Remote viewing of log events.
- The ability to set appropriate responsiveness and the frequency of updates if the log messages come at a very rapid pace.
- It is great for viewing live application log messages too because it uses a cyclical memory model, which shows the last 'X' events. Because of this, it doesn't hog a lot of memory.
- Multiple application logs can be handled by the same GUI.

log4j Performance Tips

Adding log4j statements to your code does affect performance. The following are two simple things to keep in mind that can help improve performance:

- Check if logging for a particular level is enabled before logging a message.
- As expected, simpler log formats for messages have much better performance than more elaborate custom log formats — especially those that print the date and time.

The logging performance can be improved dramatically by first determining whether a particular logging level is enabled or not. This saves the overhead associated with constructing the parameters of a logging method. This might seem like a small change; however, the parameter construction and method execution overhead for all the log messages in an application quickly add up, especially since most of them involve string concatenation.

```
if (log.isDebugEnabled()) {
    log.debug(
        "Serving content for " + contentURI
        + " from " + contentData.getContentURL()
        + ", mimeType=" + contentData.getMimeType());
}
```

When configured with the `SimpleLayout` (or a pattern using only `%p`, `%m`, or `%n`), tests have shown that logging via log4j has the same performance characteristics as the equivalent `System.out.println()` statement.

The Apache log4j developer teams provide performance figures for log4j on its Web site. For example, on an AMD Duron clocked at 800 MHz and running the JDK 1.3.1, it took about 5 nanoseconds to determine whether a logging statement should be logged or not. Actual logging was quite fast, too, ranging from 21 microseconds using `SimpleLayout` to 37 microseconds using `TTCCLayout`.

JULI

As explained earlier in the chapter, JULI is Tomcat's "container-friendly" implementation of Java Logging. This was required as Java Logging supports only one configuration per JVM, and not one per class loader. Because it is often important to be able to have a different log configuration for each Web application, the Tomcat developers chose to implement their version of Java Logging. Other than that and some extra functionality supported in the logging properties file, JULI provides the same interface to developers as Java Logging.

Java Logging Architecture

Java Logging has five components:

- Logger
- Handler
- Filter

- Level
- Formatter

Logger

These are similar to the Loggers in the log4j API, and all log requests are made to Logger objects. Again, like log4j, Loggers are arranged in a hierarchy, with child Loggers inheriting properties from their parents.

Handler

Handlers correspond to Appenders in log4j, and specify the location where log messages should be sent. A list of more useful Handlers includes:

- `ConsoleHandler`: Writes log messages to the console
- `FileHandler`: Writes log messages to a single log file or rolling log files
- `SocketHandler`: Writes log messages to a TCP socket

As with log4j, Java developers can develop custom handlers. One such custom handler is the `SMTPHandler` that functions in much the same way as log4j's `SMTPAppender`, and is available from `smtphandler.sourceforge.net`.

Level

Again, this is similar to priority levels in log4j. The difference, however, is in the actual levels themselves — seven levels (`SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, and `FINEST`) compared to the six in log4j.

There isn't a straight one-to-one correlation between log4j levels and those in Java Logging, but if you are trying to convert between the two levels, the following table provides a rough mapping.

log4j Level	Java Logging Level
`FATAL`	`SEVERE`
`ERROR`	`SEVERE`
`WARNING`	`WARNING`
`INFO`	`INFO`
`DEBUG`	`FINE`
`TRACE`	`FINER`

Finally, as with log4j, there are two special levels:

- `OFF`: The highest priority, where all logging is disabled
- `ALL`: The lowest priority, where messages at all levels are logged

Filter

Filters allow a finer degree of control over what is logged than log levels do. The Java Logging API doesn't come with any default filters, and provides this as a mechanism for developers to add custom filtering if required.

Formatter

Formatters control the formatting of information for readability as well as defined structure. Java Logging has two formatters:

- `SimpleFormatter`: Formats log messages for human readability
- `XMLFormatter`: Writes log messages in an XML structure

Formatters correspond to the layout in log4j.

A Tutorial Introduction to JULI

This section presents a simple tutorial that shows you how to get started with JULI. Because JULI is bundled with Tomcat (`<CATALINA_HOME>/bin/tomcat-juli.jar`), you don't need to copy any additional JAR files, unlike log4j.

1. Create a logging property file and pass it to your application using the `java.util.logging.config.file` system property, or programmatically. You could even add configuration into the `<CATALINA_HOME>/conf/logging.properties` file.
2. Add log statements to your application.

A sample log configuration file is shown next. As you can see, the log level is set to `FINE`, so all levels above this — `INFO`, `WARNING`, and `SEVERE` — get logged. Two Handlers are used in this configuration, namely the `ConsoleHander` and the `FileHandler`. This causes the log messages to go to the console and log file respectively. You can specify levels for the Handlers that override the level setting at the logger level.

```
# Set Handlers
handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler
# Set Handler for the 'root logger'
.handlers = java.util.logging.ConsoleHandler
# Set the level for the 'root logger'
.level = ALL
# Handler specific properties
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.pattern = %t/java%u.log
java.util.logging.FileHandler.level = WARNING
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
# Set the myPackage logger level
myPackage.level = FINE
```

Notice the pattern specified for the filename — i.e., `%t/java%u.log`. The `%t` refers to the system temporary directory (e.g., `/tmp`, `C:\tmp`) and `%u` is a unique number to ensure that there are no name conflicts. The other commonly used patterns are as follows:

- `/` is the local pathname separator.
- `%t` is the system temporary directory.
- `%h` is the value of the "`user.home`" system property.
- `%g` is the generation number to distinguish rotated logs.
- `%u` is a unique number to resolve conflicts.

Also, as you can see, the syntax does not seem to allow for multiple Handlers of the same type — i.e., more than one `FileHandler`, and so on. This is indeed a Java Logging limitation, and JULI overcomes this by allowing prefixes to be added to the loggers as shown:

```
# Set Handlers
handlers = 1java.util.logging.FileHandler, 2java.util.logging.FileHandler
# Handler specific properties
1java.util.logging.FileHandler.pattern = %t/developer-log-%u.log
1java.util.logging.FileHandler.level = FINEST
...
2java.util.logging.FileHandler.pattern = %t/sysadmin-log-%u.log
2java.util.logging.FileHandler.level = WARNING
...
```

Some of the other configurable attributes of the `FileHandler` are listed in the table that follows.

FileHandler Attribute	Significance
`java.util.logging.FileHandler.level`	Priority level, defaults to `ALL`.
`java.util.logging.FileHandler.filter`	Filter class, defaults to no filter.
`java.util.logging.FileHandler .formatter`	Name of formatter class, defaults to `XMLFormatter`.
`java.util.logging.FileHandler .encoding`	Character set encoding for logging messages, defaults to platform encoding.
`java.util.logging.FileHandler.limit`	Maximum amount in bytes to write to log file. Defaults to 0 — i.e., no limit.
`java.util.logging.FileHandler.count`	Number of log file versions in case of rolling log, defaults to 1.
`java.util.logging.FileHandler.pattern`	Pattern for output filename, defaults to `%h/java%u.log`.
`java.util.logging.FileHandler.append`	Append to file or not, defaults to `false`.

Finally, add log statements to your code. If you compare the following code to the log4j tutorial example, this usage is very similar to that in the log4j tutorial.

```
package myPackage;
import java.util.logging.Logger;
public class LoggingExample2 {
    /* Create a logger for this class */
    private static Logger log = Logger.getLogger(LoggingExample2.class.getName());
    public static void main(String[] args) {
        /* Log messages with varying severity levels */
        log.finer("This is a trace message");
        log.fine("This is a debug message");
        log.info("This is an info message");
        log.warning("This is a warning message");
        log.severe("This is a fatal/error message. All hope might be lost");
    }
}
```

In the application code, the `Logger.getLogger()` method call gets the logger object for the class. The Logger object is then used to invoke log methods to write out debug information, warnings, or error messages — e.g., `log.warning ("...")`.

Compile this class, and run it. As mentioned before, ensure that the logging property file is in the `CLASSPATH` or specified using the `Djava.util.logging.config.file=log_file` command line option.

All the logged messages appear on the console as shown:

```
Mar 18, 2007 3:27:25 PM myPackage.LoggingExample2 main
INFO: This is a info message
Mar 18, 2007 3:27:25 PM myPackage.LoggingExample2 main
WARNING: This is a warning message
Mar 18, 2007 3:27:25 PM myPackage.LoggingExample2 main
SEVERE: This is a fatal/error message. All hope might be lost
```

Now experiment with the logging levels: Move the level up from `WARN` to `ERROR` or lower it to `TRACE`, and see the amount of logging messages output to the console and the log file change.

Log Files Analysis

Log files are a rich source of information about a Web application. This information ranges from who has been using the application and when (i.e., via access logs), and what if anything is going wrong with it (i.e., via error logs).

The problem with logging is also often one of plenty. Everything logs — the database, the Web server, the Web applications themselves. What should a system administrator do with all these logs and when? Often the "when" boils down to "when something breaks down."

Thus, given the amount of data and the variety of it, there isn't one tool that can fit everyone's needs. Some of the tools that can be used to mine information from log files are presented in the following list:

- **Chainsaw** (`logging.apache.org/log4j/docs/chainsaw.html`), which was introduced earlier, is a great visual tool for viewing, querying, and tracing log data.
- **Splunk** (`www.splunk.com`) indexes log data, and provides a search engine interface for querying and navigating it.

As mentioned, log files are not just for errors:

Awstats (`awstats.sourceforge.net`) is an open source log file analyzer that can be used to determine useful information about visitors to the Web site from access logs, such as number of unique visitors, most viewed pages, and so on. There are a number of other tools with similar capabilities, including Sawmill (`www.sawmill.net/formats/tomcat.html`), Webalizer (`www.mrunix.net/webalizer/`), and WebSideStory (`www.websidestory.com/products/overview.html`) among others. To enable access logs in Tomcat, you have to configure the Access Log Valve. This is described in greater detail in Chapter 6.

While log files are an invaluable post mortem tool, they can also be a very effective tool for alerting administrators of the problem right when it occurs, so that it can be fixed with minimal customer impact. We discussed one mechanism for doing this earlier in the chapter: e-mailing critical log messages to system administrators.

Summary

This chapter introduced you to a range of logging concepts and best practices, such as maintaining *rolling logs*, having a *Nested Diagnostic Context* to distinguish between log messages from different sources, and so on. It also offered detailed coverage of both log4j as well as JULI (i.e., Tomcat's implementation of the `java.util.logging` API). To conclude this chapter, let's review some of the key points:

- The major log4j components are Logger, Appender, level, and layout.
- The major JULI components are Logger, Handler, filter, level, and formatter.
- log4j and JULI can be configured using text-based configuration files as well as programmatically.
- Individual log messages can be turned on and off based on the severity of the log message, as well as which Java package or class the log message originates from.
- Logging can be done in various formats and with custom log message patterns.
- The log messages can be sent to multiple destinations, such as the console, log files, and system loggers (for example, Unix/Linux syslog, NT Event Logger).
- Tomcat internally uses JULI for logging. The logs for Tomcat's internal messages get written to the `<TOMCAT_HOME>/logs` directory by default, and are configured using the `<TOMCAT_HOME>/conf/logging.properties` file.
- The Web applications deployed on Tomcat can use either log4j or JULI for logging, and write out their log messages to a location as per developer configuration. Ensure that you have a strategy in place to deal with the volume of log messages over time, such as by setting maximum size for the log files, setting appropriate log levels, rolling log files, and monitoring disk space usage. When a simple output format is used, and a log message is sent to the console, log4j and JULI's performance overhead is comparable to that of an equivalent `System.out.println()` statement. The flexibility afforded by using such logging frameworks in being able to change — without modifying code — what gets logged, in what format, and where is, however, incomparable.

These capabilities of log4j and JULI were demonstrated in the chapter with working examples, including code and configuration files.

20

Performance Testing

So far in this book, we have examined how to install, configure, and secure a distributed Tomcat environment. After expending the effort needed to install, configure, and tweak an installation, it is indeed a sad moment for a system administrator to see it all subverted because the application server buckles under a production load. Therefore, it is vital that administrators verify the performance characteristics of their Web applications *before* they are deployed for production use.

This chapter covers the concepts and processes of performance testing. By the end of the chapter, you will:

- ❏ Appreciate the importance of performance testing
- ❏ Understand how to develop a performance test plan
- ❏ Know how to use the Jakarta JMeter framework for performance testing and how to interpret the results

Performance Concepts

What does performance mean for a Web site? From a user's perspective, performance of a Web site boils down to how fast (or how slow) the Web pages load. However, as a Web site administrator, you need more precise ways to quantify the performance of your Web site.

What to Measure

There are two important properties that you should measure for quantifying the performance of your web site:

- ❏ Response time
- ❏ Throughput

The *response time* is the time it takes for one user to perform an operation. For example, in a storefront Web site, after the customer puts items in a shopping cart and clicks the Buy button, the time it takes to process the order and for the checkout screen to appear is the response time for the checkout Web page. Typically, you would test this operation multiple times, and note the *average response time*. Different operations on your Web site (listing items in stock, adding items to cart, checkout) might have different response time values. You can add these to get the *total transaction time* for the user. In this case, for a "Buy item" transaction, you combine the time for listing items, adding items to a cart, and checking out to get the total transaction time. Is this how long a user will have to wait every time while buying items? Not really: The response time changes with the load on the Web site.

Another performance-related property of your Web site is throughput. Throughput is the number of transactions that can occur in a given amount of time. The throughput is usually measured in transactions per second (tps).

However, before you start any performance testing, you should be clear about what kind of performance is expected from the Web site. For instance, you want to determine answers to the following questions:

- ❑ How long should a Web transaction take?
- ❑ How long should a user wait before a page loads?
- ❑ How many users should the Web site support?
- ❑ What kind of user traffic are you expecting: Is it *bursty* (meaning does it have periods of low activity and high activity)?

Understanding what kind of Web traffic is expected is important while designing performance test cases.

Scalability and Performance

The goal of load testing is to determine both the *performance* and *scalability* of a Web application. *Scalability* is the ability of a system to handle an increased load without a severe degradation of performance. Thus, the notion of scalability is related to (but distinct from) that of *performance*.

To illustrate this point, consider two fictional Web sites: Widget World and Foo Bar. Suppose that when one or two Web clients request pages from Widget World or from Foo Bar simultaneously, they receive them in an average of 250 milliseconds (ms). Thus, the two sites can be said to perform fairly well. However, when 20 Web clients request pages from the two sites, Foo Bar's performance degrades to an average of 1,800 ms per request, while Widget World continues to serve up pages at an average rate of 250 ms per request. Widget World, therefore, is considered more *scalable* (that is, it can handle scaled up loads gracefully) than Foo Bar.

The concept of scalability goes beyond designing a Web application to handle as many users as possible. Software, no matter how well written, cannot defy the laws of physics. There is a point at which even the best Web application will fail to scale up to ever-increasing demands because of the hardware limitations of the physical server upon which it runs. Therefore, scalability also reflects the capability of a Web application to maintain an acceptable level of performance when new hardware resources are added to it. Scaling a Web site by adding more hardware resources can be done in two ways: *scaling up* or *scaling out*:

- **Scaling up:** Scaling up involves moving the application to a more powerful server (faster CPU, more memory, and so on), or even adding more resources, say more memory, to an existing server.
- **Scaling out:** Scaling out involves splitting the application workload over a number of machines.

Many Web applications fail to scale because they were never designed to function across multiple servers, or they fail to take advantage of all the hardware resources available to them.

Without performance testing, administrators cannot know how scalable a Web application is, so they cannot accurately predict if a Web application will be able to perform adequately to service its anticipated load levels. Furthermore, it can tell an administrator at what point the application will fail, which is important information to know as the popularity of a Web application increases. Thus, it is very important to performance test Web applications before placing them in production.

Understanding the User's Perspective

Earlier, we discussed the user's perspective of performance. Often, one factor in this is how the user accesses the Web site. Naturally, a user accessing the Web site over a slow network connection, such as a dial-up modem, will not see the same kind of performance as a user accessing it over a high speed network, or the local LAN, will. Some features of your Web site — such as graphics-intensive Web pages, applets embedded inside a Web page, and so on — might even load so slowly on dial-up connections that the user experiences timeouts from the Web browser.

Another issue that affects performance from a user's perspective is *server proximity*. If a server is located in California, a user accessing the Web site from Korea might not see the same performance because of network-related issues, such as multiple network *hops* to reach the server.

It is important to design performance tests that simulate these different kinds of users. Many of the performance monitoring tools listed later in the chapter allow you to do this.

Measuring Performance

You can do a number of tests to determine the performance characteristics of your Web site under user load. The kinds of tests you can do fall broadly in three categories:

- Load testing
- Stress testing
- Continuous hours of operation testing

The terms "load testing" and "stress testing" are often incorrectly used interchangeably. The confusion occurs because the processes are so similar. In both tests, the system (in this case, the Web application) is subjected to multiple concurrent users. What is different in the two tests is the objective of the tests.

In *load testing*, the Web application is subjected to a "normal" amount of load. You would start with a low number of users accessing the Web application, and then increase the number of users incrementally until you reach a "high load" on the application. During this process, the response time is measured as the number of concurrent users increases. At the end of the load test, you have data on

how well the Web site scales with increased load, and if it can provide a reasonable response time at peak load.

The objective of *stress testing* is simply to break the system. You subject the Web application to an unreasonable amount of load — say 5,000 or 10,000 concurrent users for instance — and continue increasing the load until something crashes. The kind of errors you get might be the result of the Web server refusing connections, the JVM running out of memory or other resources, new database connections failing, or just about anything else. However, you are looking for the following:

- ❑ The size load that would break the system
- ❑ Bugs in the Web application that show up in such extreme conditions

The bugs that you should be concerned about relate to data corruption. For example, in a banking Web site, a crash during stress testing should not leave the bank accounts being modified at that time in an inconsistent state.

The load that the Web application is subjected to in the stress test is something out of the ordinary. Therefore, you don't have to, in most cases, go back and improve performance so that it can handle this load. All you need to do is ensure that harmful bugs (for example, data corruption) do not occur in situations of extreme load. Also, if possible, you should handle such conditions gracefully on the Web site — for example, by showing an error message asking users to try back later.

Finally, in *continuous hours of operation* (CHO) testing, you leave your Web applications running for a few days and simulate normal user traffic on your Web site. You then monitor the response time, as well as the system characteristics, including memory and CPU usage. Tools such as `top` on Linux/Unix and Task Manager on Windows can be used for this purpose.

CHO tests are useful in detecting subtle problems that are often not detected by other forms of testing. These problems could relate to resources not released properly, such as memory leaks, database connections, and/or other connection pool resources not being freed. For instance, an increase in the memory usage by the Web server or container over time might be an indication of a memory leak. This kind of testing doesn't strictly fall under the category of performance testing, however, as it is targeted toward detecting problems in the code.

Running these kinds of tests requires automated tools, as it is not feasible (or cost effective!) to get a large number of users to access your Web site manually over extended periods of time.

A lot of tools are available for performance measurement, such as commercial tools like Load Runner, and open sources ones like JMeter and The Grinder. JMeter is covered briefly later in the chapter.

After you have determined the performance characteristics for your Web site for the first time, you can use them as a *baseline*. When you make changes to the Web application for functionality or performance reasons, you can rerun your tests to see how the performance changes as compared to the baseline.

Some of the changes you make to improve performance include changing the way your Web applications work, changing how they are deployed, or even throwing more system resources at the

problem — more hardware, more memory for servers. You might also change the operating system, network, or database parameters and so on.

Before you start making changes for this, you must analyze the reasons for your performance — or the lack of it.

JMeter

Unless an administrator has a large number of people with Web browsers and a lot of spare time, special software is required to load test a Web application by simulating a heavy load.

There are numerous such load testing applications available, including open source software and commercial packages (some affordable, some extremely expensive), and some people even choose (usually in error) to write their own load tester. In this chapter, another Apache project, JMeter, is used as the load testing solution. JMeter is one of the finer solutions available. It is even capable of load testing FTP, JDBC data sources, and Java objects!

JMeter also has a lot of other features that allow you to simulate users more realistically, and thus perform better testing:

- **Timer:** A timer allows you to introduce delays between user requests. JMeter supports constant timers (a constant delay between each request), constant throughput timers (constant number of requests per minute), as well as random timers that simulate the random real-world traffic to your Web site.
- **Logic controller:** Manages the execution flow of the test plan.
- **Assertions:** Validates the data returned from the Web server.
- **HTTP proxy server:** The proxy server can record the traffic between the Web browser and the server, and this can be played back as performance test cases.
- **Distributed load testing:** JMeter can also run in a server mode, controlled by one JMeter client. This allows you to do distributed load testing, especially if you are trying to test for the effect of *server proximity* to performance.

Some of these are covered in more detail in the following sections.

Installing and Running JMeter

As of this writing, JMeter's home page is located at `http://jakarta.apache.org/jmeter/`; from there, the latest JMeter distribution can be downloaded as either a GZIP or ZIP archive. JMeter version 2.2 is used in the examples in this chapter. To install JMeter, simply extract the contents of the archive into its own directory.

JMeter runs in three modes:

- As a standalone GUI application
- Non-interactively, through the command line, or using Ant scripts
- In a server mode, for distributed testing

The standalone mode is the most common way to use JMeter. You launch JMeter by entering the `bin` directory of JMeter and running either `jmeter.bat` (on Windows) or the `jmeter` shell script (on Linux or Unix). This starts a GUI application, as shown in Figure 20-1.

The following sections demonstrate the use of JMeter. You may want to install JMeter at this point to try it out as specific features are explained.

Making and Understanding Test Plans with JMeter

Upon launching JMeter, you will see JMeter's Swing interface, as shown in Figure 20-1.

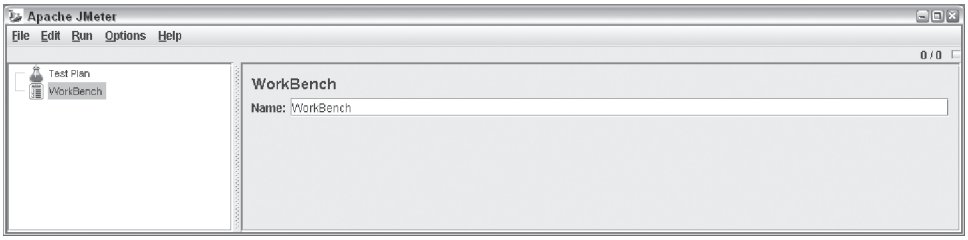

Figure 20-1: The JMeter interface

JMeter's user interface consists of a tree in the left-hand pane, representing those items and actions that you have added, and a right-hand pane that provides configuration forms and output windows for items added to the left-hand pane.

At the heart of any JMeter session is the *test plan*, which is a list of actions that JMeter will perform. The test plan is a top-level node in the test tree. Elements are added to the test plan by right-clicking on its node and selecting Add from the pop-up menu.

The second icon on the first screen, the *workbench*, is a container for test elements that are not yet part of a test plan. The workbench is a great place for experimenting with configurations and moving them back and forth between test plans.

The simplest possible test plan for testing an HTTP server will be demonstrated before the discussion moves into more advanced options in the sections to come. The first element in every test plan is a thread group. A *thread group* is a collection of elements, and each thread group has its own set of Java threads and a separate configuration.

By right-clicking the Test Plan node in the left-hand pane and selecting Add, the thread group item can be selected. After adding the thread group, its icon can be selected in the left-hand pane to expose the thread group configuration pane (see Figure 20-2). For now, the configuration options will be left at their default values, but the available options are explained.

The following configuration options are available:

- **Name:** The name doesn't need to be changed in a simple test plan, but when multiple thread groups are used, the name can come in handy to distinguish the groups.
- **Number of Threads:** This indicates the number of threads the group spawns to carry out its assigned work. Each thread is basically equivalent to an additional simultaneous user performing the tasks assigned to the group.

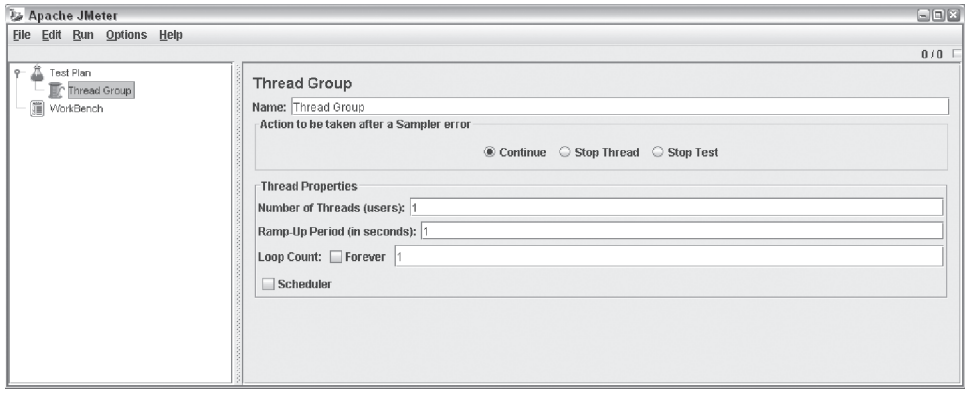

Figure 20-2: Adding a thread group

- **Ramp-Up Period (in seconds):** JMeter starts the group with one thread and add threads evenly over the course of the specified period until the value specified in Number of Threads has been reached.
- **Loop Count:** This specifies how many times the thread group will execute the elements of its assigned workload. The default is 1, which means that it will be executed once. You can also select the Forever check box, which means the elements of the test plan will be executed by the thread group until it is told to stop.
- **Scheduler:** This option enables the thread group to be configured to start and stop at a specific date/time.

Now that a thread group has been added, it's time to actually start doing something with it. Right-click the Thread Group icon to produce a pop-up menu, select Add, then Sampler, and then select HTTP Request, as shown in Figure 20-3.

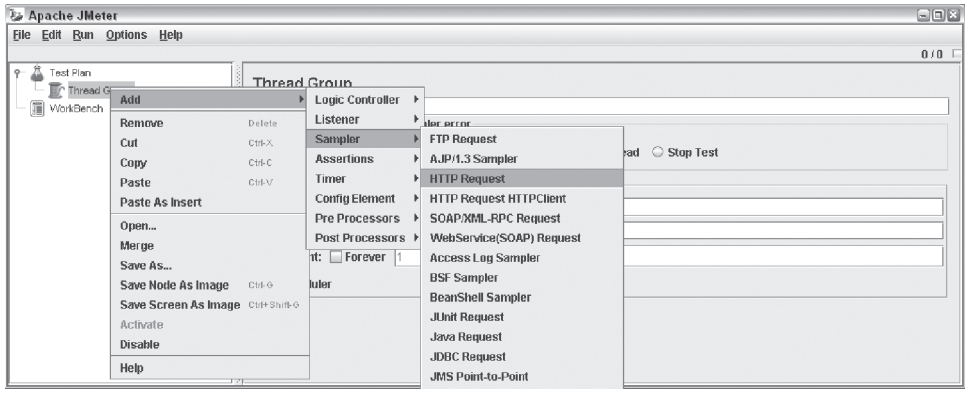

Figure 20-3: Adding an HTTP Request

Clicking the freshly added HTTP Request icon exposes its configuration panel to the right. This is a much busier screen than the thread group configuration (see Figure 20-4).

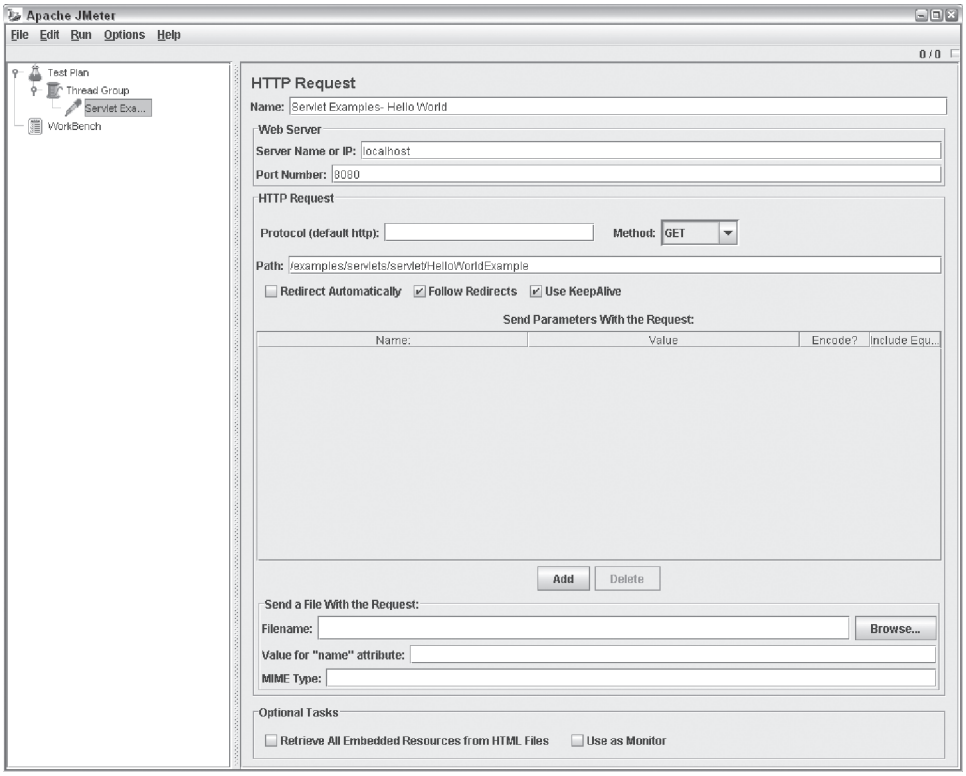

Figure 20-4: HTTP Request configuration panel

Several options are available on this screen, many of which are quite obvious. Following are explanations of some of the less intuitive options:

- **Protocol:** HTTP or HTTPS should be entered here.
- **Method:** Specifies the HTTP method to use (`GET`, `POST`, `HEAD`, `PUT`, `OPTIONS`, `DELETE`, `TRACE`). For testing a Web application, typically either a `GET` or a `POST` is selected, depending on what the requested page is expecting.
- **Path:** This is the Universal Resource Identifier (URI) of the page you are going to test. Note that `GET`-style parameters (for example, `?name=ben`) should *not* be included on this line.
- **Follow Redirects:** Web servers can return a special redirect HTTP response that instructs Web clients to request an additional URL (as opposed to HTML or some other type of response content). Web browsers typically follow these redirects in a process that is transparent to the user. This option should usually be checked.
- **Use KeepAlive:** Most Web servers and browsers support "keep-alive" connections. This type of connection is not immediately closed when a browser receives a response from the server. It is kept open for a server-configurable, short amount of time in anticipation of an additional request from the same browser. Selecting this option eliminates socket-opening latency from the load testing process, which is almost always a good idea.

- **Send Parameters with the Request:** Here's the correct place to specify `GET/POST` parameters. The `Encode?` column indicates whether the name and value should have the HTTP encoding rules applied to them. For example, characters such as the ampersand (&) or spaces need to be encoded. The `Include Equals?` column is for those rare situations in which the application is not expecting an equal sign (=) between the name and value.
- **Filename:** Some Web applications accept file uploads via HTTP `POST`. This field is for specifying which file should be uploaded with the request.
- **Value for "name" attribute:** The file will be uploaded as a key-value pair. Use this field to specify the name of the key that the Web application will use to reference the file in the request.
- **MIME Type:** This is the type of the file you are uploading. For example, an HTML file would have a MIME type `text/html`, and an Adobe Acrobat file would be `application/pdf`.
- **Retrieve All Embedded Resources from HTML Files:** Of course, Web browsers must request images and all other content referenced in an HTML page (for example, CSS pages) separately from the initial HTML page request. This option specifies whether JMeter parses the HTML and requests such resources.

Assuming Tomcat is installed on the same machine from which you are running JMeter, the server name can be set to `localhost`, the port to `8080` (the default HTTP Connector port for Tomcat), and the path to `/examples/servlets/servlet/HelloWorldExample`. This is the context path to one of the servlets supplied as a part of Tomcat examples. If JMeter were to be run on a different physical machine from the server, the server name would simply need to be set to the appropriate host or IP of the server to be load tested. All other parameters can remain unchanged for now. The completed configuration for this section should look like what is shown in Figure 20-4.

With the preceding configuration steps taken, JMeter can now be instructed to start pounding a Web application with requests for its index page. The test can be started by selecting Start from JMeter's Run menu. When you do this for the first time, JMeter asks you if you want to save the test.

However, the example isn't very practical so far, as there's no way to capture or view the results of the test. A few more options should be examined before running this first test.

JMeter was designed to internally separate the execution of a test plan from the collection and analysis of the test plan's results. For those who are interested, this is accomplished internally by use of the *Observer* or, as it is sometimes called, the *Event Listener design pattern*. This is reflected in the JMeter UI by its use of the listener terminology. *Controllers* are responsible for performing actions, and listeners are responsible for reacting to those actions. Thus, to access the results of a test plan, one of the JMeter listeners must be used.

To complete this simple test plan, you need to add a listener. This is accomplished by right-clicking the Thread Group icon, selecting Add, and then Listener. From the several built-in listeners, select the View Results Tree option.

Selecting the View Results Tree icon in the left-hand pane will expose its output window on the right. There is no configuration required for this listener. When running a test with a View Results Tree listener, you can watch each response as it is received from the server. As items are selected in the tree component of the right-hand pane, the response from the server can be viewed in the Response Data section in the bottom part of that pane.

Before starting the test, the current JMeter configuration can be saved. Right-click the Test Plan icon in the left-hand pane and choose Save As from the pop-up menu.

This first test is now ready to be run. Begin tests by selecting Start from the Run menu on the menu bar. Click the View Results Tree element before running the test. After the test is started, the Root node in the right pane's tree changes to a folder icon as test results start to trickle in. The node can be double-clicked to open it, revealing the individual test results contained within. Selecting any of the results will change the bottom pane to show the data received in the response, as well as the load time (in milliseconds), the HTTP response code, and the HTTP response message.

Figure 20-5 shows the completed test plan with the View Results Tree listener activated.

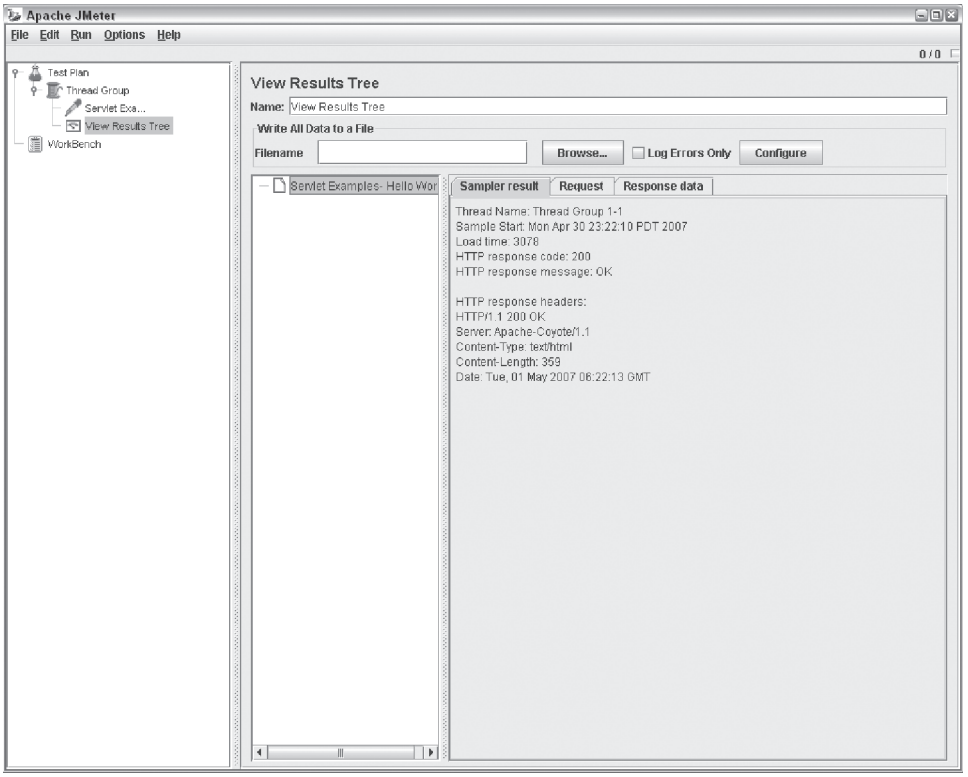

Figure 20-5: Viewing the results of a test

Because the number of threads and the loop count were both 1 by default, the test is run exactly once. You can change this on the Thread Group to a higher value for subsequent runs, say 200 threads (i.e. 200 simulated users), with a "ramp up period" of 1 second between them, running with a loop count of 2 (i.e., each request sent twice). We use this simulated load setting in some of the examples later in the chapter.

JMeter Features

Manually clicking through each result in the View Results Tree window isn't an effective way to analyze the load testing data that JMeter provides. Moreover, the simplistic mechanism used in the preceding example to generate a load is somewhat limiting. Fortunately, JMeter provides many more features to aid in capturing and analyzing load data. Following are some of the other major feature types in JMeter that will help:

- Timer
- Listener
- Logic Controller
- Sampler
- Config Element

The following sections examine the HTTP-related highlights of these feature groups.

Timer

In the sample test presented in the previous example, JMeter spawned one thread and sent one request to the Web server before ending the test. If we had selected a larger number as the loop count in the thread group, Tomcat would have fired that many requests sequentially. Real-world usage patterns for Web servers are not so simplistic. Requests sometimes arrive at the same time, and the time delay between requests is unpredictable.

To allow for more real-world simulation of request patterns, JMeter supports adding of timers to the thread group. These timers introduce logic that regulates the frequency and speed of each thread's requests. The timers supported by JMeter include:

- BeanShell timer
- Constant throughput timer
- Constant timer
- Synchronizing timer
- Gaussian random timer
- Uniform random timer

Most of these timers fall into two main categories: random timers and constant timers.

The constant timers are the *constant timer* and the *constant throughput timer*. The constant timer inserts a configurable and constant delay between each request issued by each thread in the group. The delay interval is specified in milliseconds; the default value is 300. The constant throughput timer, conversely, enables users to avoid the millisecond arithmetic and tell JMeter how many requests per minute each thread in the group should make. The default value is 60 requests (called *samples*) per minute.

The two random timers are the *uniform random timer* and the *Gaussian random timer*. These timers both simulate real-world traffic more accurately by inserting randomly calculated delays between the requests for each thread. The uniform random timer appends a truly random delay to a configurable constant delay, whereas the Gaussian random timer uses a statistical calculation to generate a pseudo-random delay. Each random timer takes a configurable constant time to which its random calculation will be appended.

The timers that don't fall into these categories (i.e., constant, random) are the *synchronizing timer* and *beanshell timer*.

The *synchronizing timer* allows to you block threads until a certain (configured) number of threads is blocked, and then release them all at once. You can thus use this timer to put a defined amount of load at a specific location in your test plan.

The last timer, the beanshell timer, allows you to write a BeanShell script for generating a delay. BeanShell is a scripting language that runs inside the JRE.

To add a timer, right-click a thread group, select Timer from the Add menu, and choose the desired timer. Timers added to a thread group will affect the entire thread group to which they are added, but will not affect peer thread groups. Adding multiple timers to a thread group will have an additive effect on the delay between requests.

Listener

As discussed previously, listeners are JMeter's way of monitoring and reacting to the results of the requests it sends. The previous example used the View Results Tree listener to show the data returned from the server, as well as the response time, the HTTP response code, and the HTTP response message. As shown previously, a listener is added by right-clicking on a thread group and selecting the desired listener from the Listener submenu under the Add menu.

The listener only listens to activity from the thread group to which it is added. For example, given two thread groups in a test plan, thread group A and thread group B, a listener added to thread group B will be oblivious to anything that happens in the scope of thread group A. The following table shows the listeners currently provided by default with JMeter.

Listener	Description
Assertion Results	Views the output of the `Assertion` elements of a thread group
Graph Full Results	A cumulative graph of the response times of each request made
Graph Results	A simple graph view, plotting individual data points, mean response time, and standard deviation of response time for all requests in its parent thread group
Simple Data Writer	Writes the URLs sampled and their associated response times to a file for further analysis or posterity
View Results in Table	Provides a real-time view of test results organized into a table
View Results Tree	Provides a real-time view of test results organized into a tree
Aggregate Report	Displays aggregate information about each resource requested, such as the number of requests, the average response time, and so on
Spline Visualizer	A graph view of all data points made during a test plan run. The results are shown as an interpolated curve.

Each listener can be grouped into one of the following categories:

- ❑ Visualization listeners
- ❑ Data listeners
- ❑ Other listeners

Visualization Listeners

Graph Full Results, Graph Results, and Spline Visualizer all create graphical, real-time depictions of the test results. Graph Results (shown in Figure 20-6) is the simplest and most popular of these, plotting average response time in blue, standard deviation in red, median in purple, throughput in green, and each individual data point in black. In the graph shown, the median and throughput graphs are hidden; the other values are not shown in color in this book, but the standard deviation is the upper line, the average response time is the lower line, and the data points are the dots.

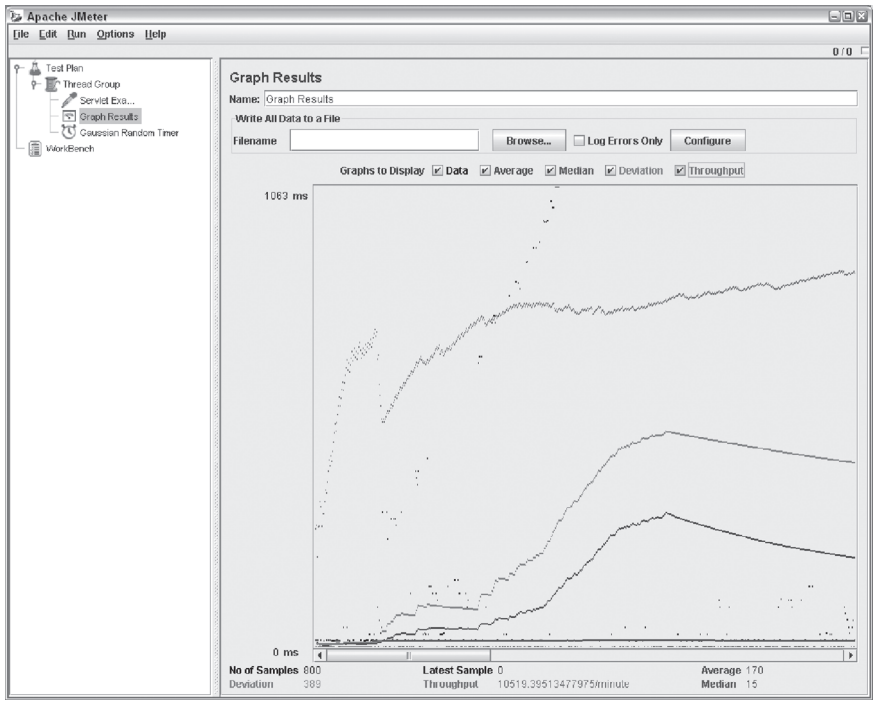

Figure 20-6: Example of a Graph Results listener

More information about how to make sense of this data is presented in the section "Interpreting Test Results," later in this chapter.

Data Listeners

The data listeners include Simple Data Writer, View Results in Table, and View Results Tree. These listeners capture the raw data, response time, and return codes from the server. The Simple Data Writer is somewhat redundant, as all of the other listener tools enable the raw data to be saved to a file. However, if no other listener is used, it can be useful to use Simple Data Writer by itself to record the data. Having the raw data is an important tool, because it enables users to keep their data for their records, as well as to potentially import the data into other tools for more detailed analysis.

You can even write the test data out to a file, by selecting the file in the Write All Data to a File section (see Figure 20-6). By default, the file is in a CSV format, and you can choose to save it as XML using the Configure option button.

A sample of the data generated is shown here:

```
<testResults version="1.2">
   <httpSample t="0" lt="0" ts="1178001101546" s="true" lb="Servlet Examples- Hello
World" rc="200" rm="OK" tn="Thread Group 1-14" dt="text"/>
   <httpSample t="0" lt="0" ts="1178001101609" s="true" lb="Servlet Examples- Hello
World" rc="200" rm="OK" tn="Thread Group 1-1" dt="text"/>
   ...
</testResults>
```

The XML format is fairly intuitive:

- `t`: Elapsed time for the request
- `lt`: Latency
- `ts`: Timestamp as number of milliseconds since Jan 1, 1970
- `s`: Success (true/false)
- `lb`: Label
- `rc`: HTTP Response code
- `rm`: Response message
- `tn`: Thread group name
- `dt`: Data encoding

Aggregate Report, another data listener (see Figure 20-7), does more than display raw data. It organizes the raw data by requested URL, and provides a summary of all the data points involving that URL. It is a useful and concise way to track performance, striking a balance between the graphical visualizations and the other raw data listeners.

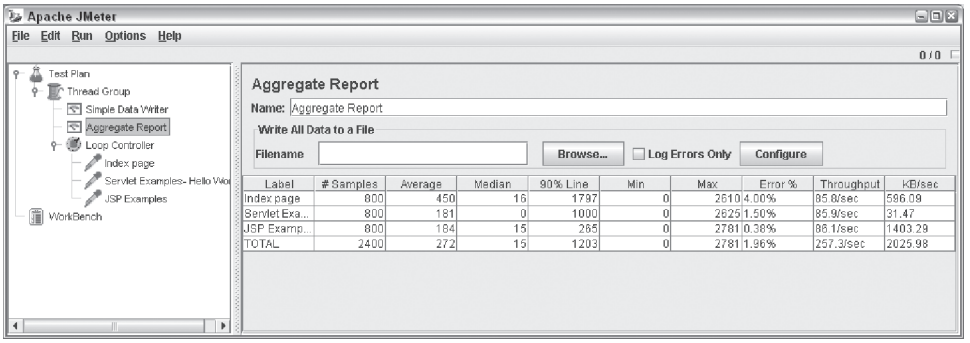
Apache JMeter
File Edit Run Options Help
0 / 0
Test Plan
Thread Group
Simple Data Writer
Aggregate Report
Loop Controller
Index page
Servlet Examples- Hello Wo
JSP Examples
WorkBench
Aggregate Report
Name: Aggregate Report
Write All Data to a File
Filename
Browse...
Log Errors Only
Configure
Label # Samples Average Median 90% Line Min Max Error % Throughput KB/sec
Index page 800 450 16 1797 0 2610 4.00% 85.8/sec 586.09
Servlet Exa... 800 181 0 1000 0 2625 1.50% 85.9/sec 31.47
JSP Examp... 800 184 15 265 0 2781 0.38% 86.1/sec 1403.29
TOTAL 2400 272 15 1203 0 2781 1.96% 257.3/sec 2025.98

Figure 20-7: Aggregate Report makes the results of a test easy to interpret

Assertion Results

Assertion Results enables users to view the results of `Assertion` elements that have been added to samplers. Both assertion and assertion results are discussed in the section "Sampler," later in this chapter.

Logic Controller

A Logic Controller's primary purpose is to manage the execution flow of a test plan. It is a container for other executable test plan elements. Logic Controllers that are added to a thread group (or even as a subnode of another Logic Controller) will be treated by their parent execution context as a single node to be executed. Elements added beneath logic controller nodes will be executed according to the rules of the specific Logic Controller to which they are added.

Like thread groups, Logic Controllers create a separate visibility space for listeners, timers, and other elements that are context-specific. Logic Controllers can be thought of as the closest approximation JMeter test plans have to the `while`, `for`, and `function` constructs of typical programming languages.

The following sections discuss the built-in Logic Controllers that currently ship with JMeter:

- ❏ Interleave Controller
- ❏ Switch Controller
- ❏ Simple Controller
- ❏ Loop Controller
- ❏ If Controller
- ❏ While Controller
- ❏ ForEach Controller
- ❏ Include Controller
- ❏ Module Controller
- ❏ Once Only Controller
- ❏ Random Controller
- ❏ Random Order Controller
- ❏ Runtime Controller
- ❏ Throughput Controller
- ❏ Transaction Controller
- ❏ Recording Controller

Some of these Logic controllers are explained in more detail in the following sections.

Interleave Controller

The Interleave Controller executes one of its subelements each time its parent container loops. It executes subelements in the order in which they are listed in the configuration tree. For example, if a user were to create an Interleave Controller with four elements under a thread group set to loop 14 times, JMeter would execute the entire set of Interleave Controller subelements three times, and would then execute only the first two subelements a fourth time (4 + 4 + 4 + 2 = 14).

Interleave Controllers are good for testing a sequential process in which each request depends on the successful completion of the previous request. An obvious example is an online shopping application,

whereby a user searches for an item, adds it to a shopping cart, enters credit card details, and finalizes the order.

Switch Controller

The Switch Controller acts like the Interleave Controller in that it runs one of the subelements on each iteration. However, rather than run them in sequence, the controller runs the element number defined by the switch value.

Simple Controller

With the Simple Controller, each subelement is executed each time the thread group loops. The Simple Controller can be used to logically organize test elements in much the same way as folders are used on a file system to logically separate their contents. If a site were to be load tested with a nontrivial amount of functionality, it would make sense to use Simple Controller elements to separate the tested functionality into related modules to keep the test plan more maintainable. This enhances the maintainability of the test plan in the same way that dividing large software projects into modules and functions enhances maintainability of the software.

Loop Controller

The Loop Controller loops through all of its subelements as many times as specified in the Loop Controller's configuration panel. Therefore, any elements under the Loop Controller will execute the specified number of times, multiplied by the number of times the parent thread is set to loop. If a Loop Controller were configured to loop four times under a thread group that loops four times, each subelement of the loop controller would be executed 16 times.

If Controller

The If Controller allows JMeter to control whether the subelements below it are allowed to run or not. This decision is made on the basis of a conditional statement.

While Controller

The While Controller loops through all subelements until the condition specified is false.

Module Controller

The Module Controller enables users to insert elements of the test plan *from entirely different locations* into the context of the module controller. In other words, the Module Controller can be thought of as a way to execute a method (or function or subroutine, depending on your preferred lexicon). Module Controllers can be particularly useful for applying to testing all of those reuse principles that programmers espouse. The Module Controller can also be used to radically design a test without physically moving testing elements around. Figure 20-8 shows the Module Controller.

Once Only Controller

Not surprisingly, the Once Only Controller executes its child elements only once during the run of a load test. This controller can be used to execute an initial login, to create an application entity on which other tests depend (for example, creating an order in a sales application so you can manipulate it with other requests), or to perform any other operation that needs to happen only once.

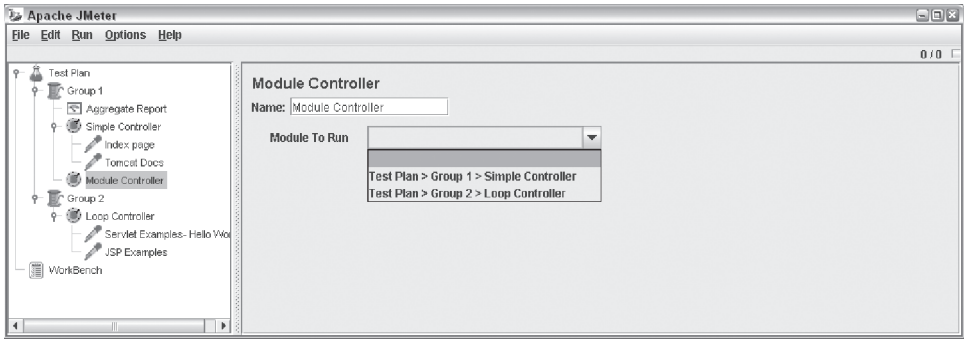

Figure 20-8: Configuring a Module Controller

Random Controller

The Random Controller works just like the Interleave Controller, with one exception. Whereas the Interleave Controller executes one item from its list in sequential order, the Random Controller picks an item in its collection of subelements at random each time it is executed.

Throughput Controller

The Throughput Controller provides another mechanism for testing creators for throttling the number of requests sent. The Throughput Controller will send only a subset of the requests sent to it on to its subelements. This subset can be defined in terms of *total executions* or *percent executions*. For example, the Throughput Controller can be configured to send only 50 percent of its requested executions on to its subelements, or it can be configured to send only the first 10 requests on to its subelements. Thus, the Throughput Controller could be configured to emulate the Once Only Controller mentioned previously.

The Throughput Controller can be configured to limit *all* of the threads in a group collectively, or it can be set to limit them individually. This is controlled by selecting the Per User check box. When it is on, each thread in the group is individually subject to the throughput control. Otherwise, the controls apply to the group.

Recording Controller

The Recording Controller is used much differently from the other controllers. The Recording Controller is used by the *HTTP Proxy Server* feature of JMeter. The HTTP Proxy Server enables JMeter to record requests made by a Web browser and incorporate them as part of a test plan. The HTTP Proxy Server uses the Recording Controller to save the data it receives.

Sampler

As mentioned in our first JMeter example, samplers generate requests to be sent to a server. Following are some of the commonly used samplers:

- ❑ FTP Request
- ❑ HTTP Request
- ❑ WebService Request (SOAP, XML-RPC)
- ❑ Java Request

- ❏ JDBC Request
- ❏ JMS Request (Point to Point, Publisher, and Subscriber)
- ❏ JUnit Request
- ❏ LDAP Request

As shown, JMeter can be used to load test more than just Web servers. A variety of test plans can be created by using some of these sampler types. While all of these sampler types are interesting, this chapter focuses on the HTTP Request sampler. The basic parameters of HTTP requests were previously discussed. This section addresses some of the advanced configuration options of the HTTP Request sampler.

Config Elements

Config Elements enable various configurable attributes to be applied globally to a series of samplers. In the case of HTTP requests, four different Config Elements can be used:

- ❏ HTTP Header Manager
- ❏ HTTP Authorization Manager
- ❏ HTTP Cookie Manager
- ❏ HTTP Request Defaults

While Config Elements are normally added to thread groups or other containers for samplers, they can also be added to individual samplers to override global values.

HTTP Header Manager

In some cases, application testing requires specific HTTP headers to be set in order to get a valid reflection of true application performance. For example, if an application performs different actions depending on the browser type making the request, it is necessary to set the `User-Agent` header when making test requests. The HTTP Header Manager is used to explicitly set header keys and values to be sent as part of each request. If added as a node under an HTTP Request element, the custom headers will be sent only for the request under which they are added. These headers will be sent with every request in the same branch if they are set at the thread group level.

Configuring the HTTP Header Manager is simple and very similar to configuring the `Name/Value` parameters in an `HTTP Request` element.

HTTP Authorization Manager

The HTTP Authorization Manager handles requests that require HTTP authentication. Like the HTTP Header Managers, they can be added either directly under an `HTTP Request` element or to an entire branch of a tree. Their configuration parameters are simple, accepting a base URL from which they will attempt to send authentication credentials, plus the obligatory username and password.

HTTP Cookie Manager

Many modern Web applications use cookies in some manner. In these cases, an HTTP Cookie Manager element will need to be added to the test plan. Like HTTP Authorization Managers and HTTP Header

Managers, HTTP Cookie Managers can accept a hard-coded list of cookies that should be sent for every request. In this way, a test sampler can emulate a browser that has previously visited a site. Additionally, HTTP Cookie Managers can mimic a browser's ability to receive, store, and resend cookies. Therefore, for example, if a cookie is dynamically assigned to each visitor, the HTTP Cookie Manager will receive it and resend it with every appropriate subsequent request.

HTTP Cookie Managers can be added either to a thread group or directly to an `HTTP Request` element, depending on the scope of their intended influence.

Note that the HTTP Cookie Manager stores cookies on a per-thread (also called *per-user* in JMeter) basis. That is, given a thread group of 10 threads, if each thread receives a different cookie for the same base URL, each thread will continue to resend that unique cookie. However, when cookies are added manually to the Cookie Manager, *all* threads will resend that cookie. Future versions of JMeter may correct this limitation.

HTTP Request Defaults

HTTP Request Defaults provide a convenient mechanism for using common values among unique HTTP request samplers. The basic request parameters (such as `protocol`, `host`, `port`, `path`, and `name/value` parameters) can be specified in one location to be shared among the samplers.

Assertions

Even if an application is responding lightning fast, there is no cause for celebration if its output is invalid. An *assertion* provides a way to validate the actual data returned as a result of each request so that users can be sure that the server is both responsive *and* reliable. Assertions are created as subelements of samplers (such as the HTTP Request sampler). An assertion is a declaration of some truth that should be verified.

For example, you could create an assertion that posits that the response from a server should contain the word `Hello`. When this assertion exists, the response of the HTTP request to which it is added will be checked for the existence of `Hello`, and will throw an assertion failure if the string is not present.

Using the following simple HTML file, we'll see how an assertion is used to validate it:

```
<html>
  <body>
    Hello, World!
  </body>
</html>
```

Given that the preceding HTML has already deployed to an accessible HTTP server, the first step is to add an HTTP Request sampler into a test plan. After creating the HTTP Request sampler, it can be selected and right-clicked. From the context menu, choose Assertions➪Response Assertion. Clicking the new Response Assertion icon will display a panel like the one shown in Figure 20-9.

In Figure 20-9, the text "Hello World" has already been added to the assertion. This is accomplished by clicking the Add button and writing the text into the entry area that appears. In this example, the Contains option has been selected. This indicates that the assertion will verify that the response from the HTTP request contains the pattern "Hello World!" When the Matches option is selected, the entire response must match the pattern.

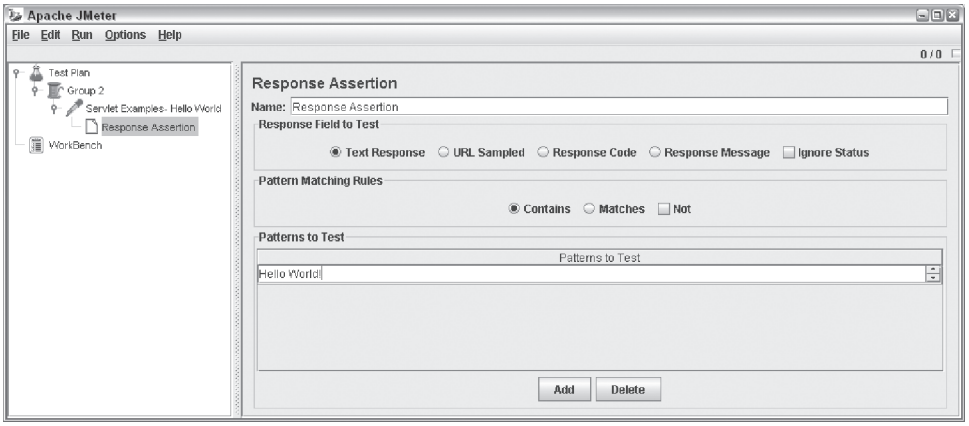

Figure 20-9: The Response Assertion configuration screen

Now that an assertion has been added, an Assertion Results listener (mentioned previously) will need to be added to view the successes and failures of the assertion. The Assertion Results listener should be added to the parent thread group of the HTTP Request sampler. Assertion Results listeners require no configuration. They display all assertion results for any assertion in their thread group. If the assertion passes, it prints the string identifying the resource request (in this case, the URL). If there is a failure, it prints an indented failure message directly below the resource identifier, stating the pattern match that failed.

The Response Assertion can be used to verify that a response URL or body contains some pattern of text. Some of the other assertion types include the following:

- **Duration Assertion:** Verifies that a response occurs within a specified time period.
- **Size Assertion:** Checks the size of the response, in bytes, and verifies that it is equal to, not equal to, greater than, less than, greater than or equal to, or less than or equal to a specified value.
- **XML Assertion:** Ensures that the response is well-formed XML. It does not check validity against a Document Type Definition (DTD) or other schema type.

These assertions can be added to a test plan and configured in the same manner as the Response Assertion.

HTTP Proxy Server

Creating HTTP Request elements can get tedious. JMeter provides an HTTP Proxy Server that enables it to monitor browser activity and auto-generate HTTP Request elements based on the requests made from the Web browser. As depicted in Figure 20-10, the proxy works by intercepting all requests that are made during the browsing session, converting them into JMeter HTTP Request elements, and passing the request on to the destination Web server.

Generating a large number of HTTP Request elements for a test plan is as simple as reconfiguring a browser's HTTP proxy setting and clicking through the features that should be load tested.

To start using the HTTP Proxy Server, it must be added to the JMeter WorkBench. This is done by right-clicking the WorkBench icon and selecting Add➪Non-Test Elements➪HTTP Proxy Server. The proxy's configuration screen is shown in Figure 20-11. Note also the Recording Controller defined in the

WorkBench. This is set as the Target Controller in the HTTP Proxy Server. When you run requests through the proxy server, they get recorded under this Recording Controller.

Note the port number (default 8080) has been changed to 9090.

If Tomcat is running on the same machine as JMeter and is using its default port of 8080, the proxy's default port will conflict. On such machines, JMeter's proxy port will need to be changed to an unused port on the system.

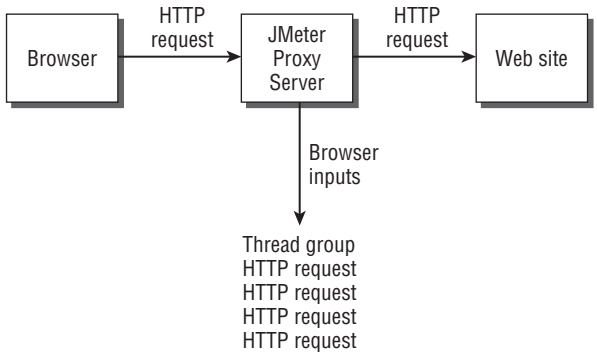

Figure 20-10: The JMeter Proxy Server sits in between a browser and a Web site.

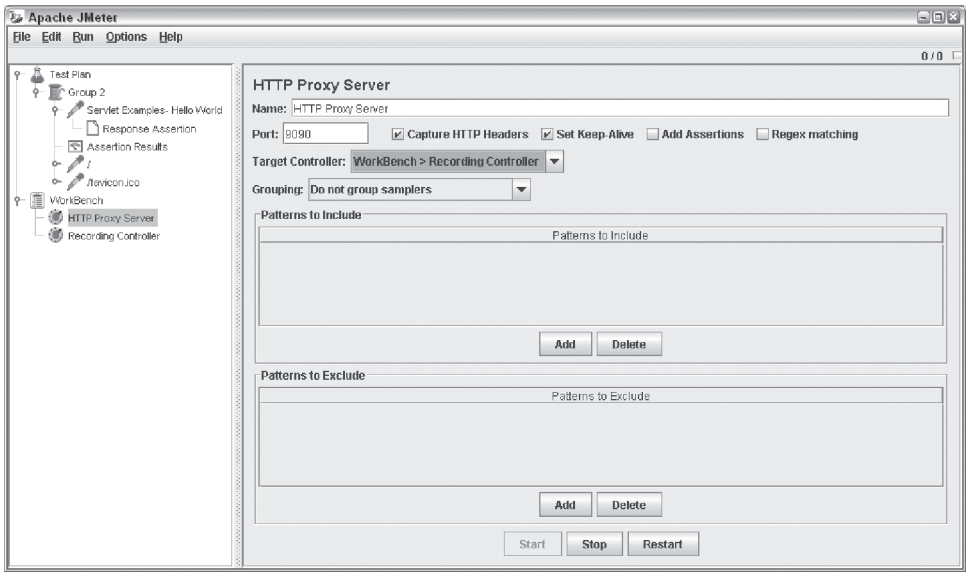

Figure 20-11: Configuration screen of the HTTP Proxy Server

In the Patterns to Include and Patterns to Exclude text boxes, regular expressions can optionally be added that match the URLs for which HTTP Request elements will be generated. All browser traffic will be proxied through the HTTP Proxy element. However, if any patterns are listed here, only requests that match an include pattern or are not excluded by a pattern will be converted into HTTP Request elements.

The following instructions will aid in configuring browsers to use a proxy server:

- **Internet Explorer 6:** Select the Tools➪Internet Options pull-down menu. Now select the Connections and click the LAN Settings button. Select the Use a proxy server for your LAN check box. In the Address field, type **localhost** (or the host name of the computer running the JMeter HTTP proxy — it need not be the local machine), and in the Port field, type the port chosen for the JMeter proxy. Finally, clear the Bypass proxy server for local address check box.
- **Firefox 2:** Bring up the Preferences dialog box by choosing Tools➪Options. Select the Advanced icon at the top, and then the Network tab element. Click on the Settings button, and select Manual proxy configuration. Type **localhost** in the HTTP Proxy field, and the proxy port configured in JMeter in the Port field. Note that any HTTPS requests will need to be entered in the SSL Proxy field. Also, remove localhost from the No proxy for list.

After configuring the proxy server, it must be started by clicking the Start button. Now start your browser and run through your tests using it. Because you have set JMeter as the proxy, any request to the Web server will be sent via the JMeter Proxy Server. These requests will then be captured by JMeter and converted into HTTP Request items in WorkBench under the Recording Controller. These items can then be dragged into the test plan.

Distributed Load Testing

JMeter has the capability to coordinate multiple JMeter instances distributed across a network, instructing them to perform the same load test. To enable this functionality, one or more JMeter instances starts in *server mode*. Server mode enables a JMeter client (that is, the normal JMeter GUI interface) to attach to the JMeter instance in server mode and control it. The JMeter client's listeners receive the data generated by the JMeter server mode instances. There are two potential use cases for this capability:

- **Scalability:** For various reasons, a single JMeter instance might not be enough to generate a load heavy enough to bring a Web application to its knees. In these rare circumstances, JMeter's distributed capability can be used to generate a truly massive barrage of requests. This capability could also be used to coordinate a massive distributed denial of service (DDoS) attack, which highlights a good point: *Don't ever load test a Web site that you don't control!*
- **Server Proximity:** As an extreme example, imagine trying to load test a server in the Eastern United States while sitting in India. It would be difficult to have any acceptable degree of confidence in the tests, as network latency and packet loss would play such a critical role in the process. In a case such as this, it's possible to install a JMeter server somewhere physically close to the server being tested, and then use that server to perform the tests remotely — unless off course you meant to test for real, random network latency in the first place.

Figure 20-12 illustrates the method by which JMeter interacts remotely between a JMeter client, a JMeter server, and a Web server.

Figure 20-12: One JMeter client can control multiple JMeter servers.

The following is a step-by-step guide to setting up and running tests using the JMeter server:

1. Install the standard JMeter distribution on the intended JMeter server machine(s).
2. For each JMeter instance to be controlled, start the JMeter server process by entering JMeter's `bin` directory and executing *either* `jmeter-server.bat` or the `jmeter-server` shell script, depending on your host operating system. An error message seen sometimes on Windows is "Windows cannot find rmiregistery." If you get this error, add `%JAVA_HOME%/bin` to your system `PATH`.
3. On the client machine, edit the `jmeter.properties` file (located in the `bin` directory of JMeter's installation directory).
4. Uncomment and change the `remote_hosts` property to include a comma-separated list of the JMeter server machines. These can be either DNS-resolvable host names or IP addresses.
5. Start the JMeter client by running `jmeter.bat` or the `jmeter` shell script.
6. Load or create a new test plan.
7. Two new options will be present in the Run menu: Remote Start and Remote Stop. In each of these options, any servers listed in the `remote_hosts` property will be available for stopping and starting.
8. Proceed with the JMeter GUI as usual.

Interpreting Test Results

Generating reams of raw data is one thing; knowing what the data means is quite another. There are two basic ways to analyze the resulting data:

- ❏ Set performance goals and test them.
- ❏ Establish the scalability limitations.

Setting Goals and Testing Them

A particularly effective way to create a Web application is to establish up front the performance goals for the system. These goals might include one or more of the following metrics:

- ❏ What's the average amount of time a user should wait for a request to be fulfilled?
- ❏ What's the longest amount of time a user should be made to wait?
- ❏ How many concurrent requests should be supported before errors occur?

Once such goals are established, it is a simple matter to determine if the tested Web application lives up to them. Both the *Aggregate Report* and the *Graph Results* listeners are perfectly adequate for such an analysis.

As an example, consider the following hypothetical scenario: A simple Web site must support five concurrent users, where concurrent is defined as users making requests no more than a second apart. Each user should receive a response in an average of 250 milliseconds, although outlying delays up to 3 seconds are acceptable.

To continue the scenario, suppose a load test were created that measured the application using the criteria established in the goals. Figure 20-13 shows the Aggregate Report created by the test plan.

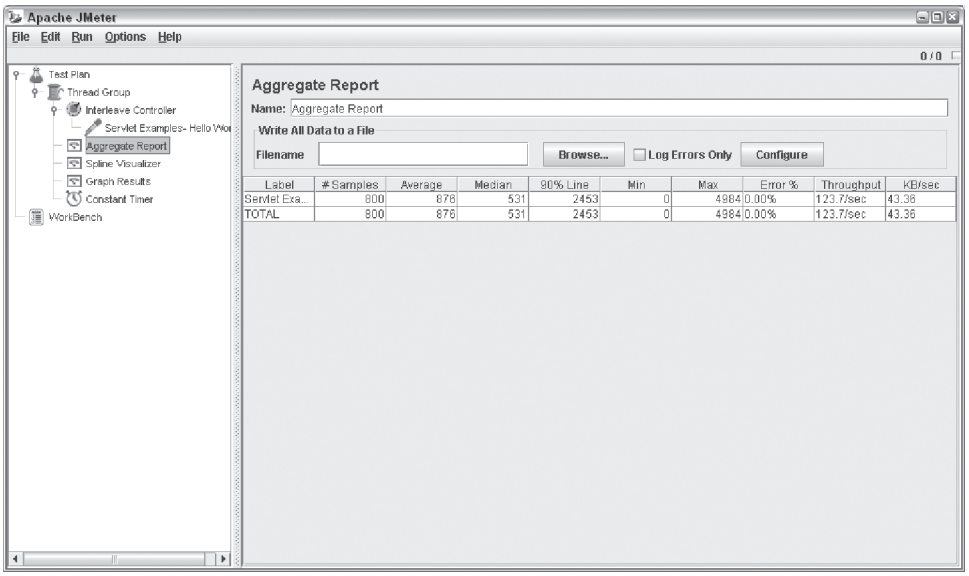
Apache JMeter
File Edit Run Options Help
0 / 0
Test Plan
Thread Group
Interleave Controller
Servlet Examples- Hello Wor
Aggregate Report
Spline Visualizer
Graph Results
Constant Timer
WorkBench
Aggregate Report
Name: Aggregate Report
Write All Data to a File
Filename
Browse...
Log Errors Only
Configure
Label | # Samples | Average | Median | 90% Line | Min | Max | Error % | Throughput | KB/sec
Servlet Exa... | 800 | 876 | 531 | 2453 | 0 | 4984 | 0.00% | 123.7/sec | 43.36
TOTAL | 800 | 876 | 531 | 2453 | 0 | 4984 | 0.00% | 123.7/sec | 43.36

Figure 20-13: The example scenario's Aggregate Report screen

With the Aggregrate Report's presentation of the data, it is clear that the Web application did not meet its goals. The average delay was 876 milliseconds (ms), roughly about three times worse than the goal. The outlying delay was 4984 ms, or almost 5 seconds. However, on a positive note, no requests resulted in an error from the Web application.

The Graph Results listener can provide additional valuable information (such as the true distribution of the responses), as shown in Figure 20-14.

Note that all the graphs except Data have been disabled. This makes the graph easier to interpret in black and white. The dots on the graph indicate each individual sample, and how long the response took in milliseconds. Higher dots represent longer delays. Thus, the graph provides greater granularity regarding the application's performance. You can clearly see that the application's performance wasn't consistent at all!

A separate statistic, the *deviation*, confirms this observation. Deviation (commonly called the *standard deviation*) is a measure of how widely values in a sample are dispersed from the average value. A higher standard deviation indicates more variance in response time.

The graph also provides a new statistic: the *median*. If all the response times were placed in a sorted list, the median is the number in the middle.

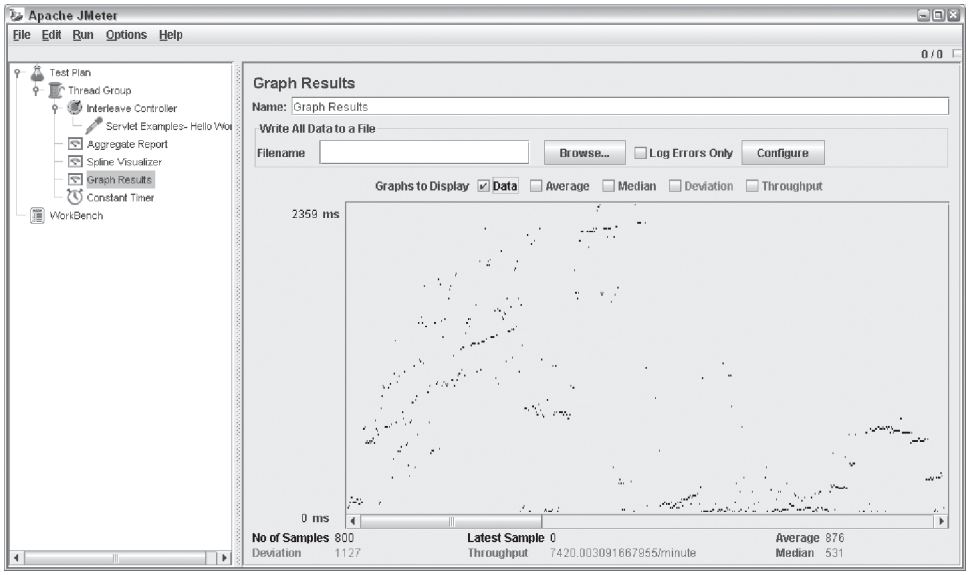

Figure 20-14: Graph Results output for the example scenario

The throughput number provided in the graph is the same value as the "rate" in the *Aggregate Report*. However, it is given at a different scale (minutes instead of seconds).

Establishing Scalability Limitations

Sometimes, performance goals aren't important, and the only real motivating factor is determining the point at which a Web application breaks.

Here, JMeter comes in very handy, especially with its distributed testing capabilities. It is important to remember, however, that even these stress tests should be configured in such a way that they are representative of real-world scenarios. It is not very useful to know that if 10,000 users make concurrent requests for the same URL every 10 ms, the tested Web application fails. Such a scenario is unlikely to occur in the real world. It is more helpful to know that if 500 simultaneous users follow 1 of 10 realistic sequences of requests (averaging a 3-second delay between each request), the Web application will generate errors for 4 percent of the responses.

When performing such stress tests, the Aggregate Report can handily show the error rate, and it can be configured to log all errors that occur, for further analysis.

Further Analysis

JMeter can, of course, save all of its performance data for analysis by other tools. Although at this time there are no (well-known) performance analysis tools designed specifically with JMeter in mind, popular programs such as Microsoft Excel can import JMeter data and generate all sorts of handy reports and graphs. By default, JMeter can record output in either XML or *Comma Separated Value* (CSV) format. This can be configured by setting the `jmeter.save.saveservice.output_format` property in `jmeter.properties` to either *xml* or *csv*. The `jmeter.properties` file is located under `<JMETER_INSTALL_DIRECTORY>/bin`.

Alternatives to JMeter

Although this chapter's discussion focused on JMeter for load testing, numerous other tools are available — those in the public domain and as well as commercial products.

Some popular tools are listed in the following table.

Performance tool	Company	URL	Licensing
Flood	Apache Software Foundation	`http://httpd.apache.org/test/flood/`	Open source, Apache license
The Grinder	—	`http://grinder.sourceforge.net`	Open source, BSD license
LoadRunner	Hewlett-Packard	`mercury.com`	Commercial
SilkPerformer	Borland	`borland.com/us/products/silk/silkperformer/index.html`	Commercial
Rational Performance Tester	IBM	`www-306.ibm.com/software/awdtools/tester/performance/`	Commercial
WebLOAD	Radview Software	`radview.com`	Commercial

Two of the market leaders are Borland's SilkPerformer and Hewlett-Packard's LoadRunner. Both offer very advanced scripting features, thus providing end users with great flexibility in specifying load test behaviors. With this flexibility comes the price of increased complexity, but both companies offer comprehensive documentation as well as training programs for their products.

The open source world also has more to offer in terms of load testing tools. From the up-and-coming OpenLoad (`http://openwebload.sourceforge.net/`) to the more mature Grinder (`http://grinder.sourceforge.net`), numerous options are available. A great resource for finding the latest in open source load testing tools is the FreshMeat open source software archive site (`http://freshmeat.net/browse/`), which includes a category specifically for these kinds of tools under Topic➪Software Development➪Testing➪Traffic Generation.

Additionally, many other testing resources are available on the Web. A great place to look for an ever-evolving list of sites is the Open Directory Software Testing list (`www.dmoz.org/Computers/Programming/Software_Testing/`), which is a directory of sites that focus on testing topics.

What to Do After Performance Testing

After you complete the initial performance testing, and find that your Web application is very slow or cannot handle the user load, you might be tempted to rush in with a solution, redo parts that seem slow, throw more hardware at the problem, and so on. Don't!

It is important to analyze the reasons your application is slow and to identify the bottlenecks. This is a complex task, and would require involvement of both system administrators as well as the developers of the Web application.

Some of the steps that you can perform to identify the root causes of your performance problem are listed here:

- Log your system information, such as the memory and CPU utilization on your server. You can do this by using tools such as top, iostat, and vmstat on Linux and the Task Manager and PerfMon on Windows.
- Database-related issues are often a major cause of performance bottlenecks. Use the diagnostic tools, if any, provided by your database vendor. P6Spy (`www.p6spy.com`) is an open source tool that helps in timing database calls.
- *Instrument* the Web application code to trace the areas of poor application performance. Instrumenting the Web application for performance analysis means adding code to track and record the amount of time it takes to perform certain tasks. Typically, this is in the form of logging statements that print out the time taken for operations. For example, developers would add such statements around database invocations, calls to remote methods, and so on. In Chapter 19, you saw how logging statements can be added to Web applications.
- Use profiling tools to analyze your Web applications, draw visual representations of the behavior under load, and show call graphs with the time taken in each module. The call graph is a diagram that identifies the modules of a system, and shows which module calls another. Some popular profiling tools for Web applications include commercial ones such as JProbe (`www.quest.com/jprobe/`) and OptimizeIt (`www.borland.com/optimizeit/`), and free ones such as the Eclipse "Test and Performance Tools Platform" (`www.eclipse.org/tptp/`).

The objective of using these measures is to determine what the performance bottlenecks in your application are. Once you understand what the bottlenecks are, you can proceed to address them specifically.

Summary

Performance testing is an important but often overlooked activity. Besides enabling your applications to scale, it also helps in making decisions about system architecture, and to validate decisions made previously. This chapter covered the following material:

- You should decide what you are going to measure, and develop a performance test plan before you start. The test plan typically includes the desired performance (how many concurrent users to support, expected response time, expected throughput), as well as the testing methodology and their details.
- The test plan should also include the user transactions being simulated, the load being tested for, and the environment for testing.
- Scalability (a system's capability to handle increased load without experiencing degraded performance or reliability) is driven by many factors. In a Tomcat installation, some of those factors are server hardware, software configuration, and deployment architecture.

- JMeter is a full-featured, open source load tester, which is part of the Jakarta project. Various JMeter load testing techniques were explored.
- After performance testing is completed, analysis of the test results helps determine the root causes of poor performance. This should be done before attempting any changes for performance.

The next chapter covers diagnosis of common performance issues and Tomcat-specific performance tuning in more detail.

21

Performance Tuning

The previous chapter ended with a section called "What to Do After Performance Testing." It advised you to first analyze the root cause of poor performance before attempting a fix. This chapter covers some of those steps in more detail.

- First, we discuss general best practices for approaching a performance tuning project, such as setting up a test bed, establishing a performance *baseline*, making incremental changes for tuning performance, and measuring performance changes, if any, against this baseline.
- Next, we suggest where to look for performance bottlenecks, and how to identify the root cause when faced with specific Tomcat issues.
- Finally, we list design and development choices, and configuration options that affect performance in Web applications in general, and Tomcat-hosted applications in particular. Some of these tips might be common knowledge among Web developers, and others are mentioned elsewhere in the book. This section collects these tips in one place.

Performance tuning is a complex area, and the factors that affect performance are sometimes outside the application itself. These factors may include the network configuration and hardware platform characteristics, the operating system settings, Java virtual machine parameters, database tuning parameters, and finally the architecture of the Web application itself. Naturally, everything that affects performance cannot be covered here. The objective of this chapter is twofold: first to give you a *feel* for how to approach performance tuning in general; and second, to explain how to go about tuning the Tomcat configuration and your Web application for improved performance.

Performance Tuning Best Practices

Performance tuning should not be approached in an ad hoc fashion. Performing a number of fixes in your application all at once can make it difficult to understand what caused the performance difference. Worse, you might just cause another problem, or undo the improvements made earlier. A better approach is an iterative one, and an approach that applies just one fix at a time. Thus, the recommended approach is:

1. Set up a test bed.
2. Do performance testing. Use the results of your first test run as a performance benchmark or baseline for subsequent testing.
3. Investigate performance bottlenecks in your application, deployment environment, and other systems that it interacts with.
4. Identify areas where improvements are needed; these might involve implementation changes as well as deployment or configuration tuning of your Web application.
5. Make one fix.
6. Repeat steps 2 through 5.

When should you think about your application's performance?

A quote attributed to Donald Knuth says, "Premature optimization is the root of all evil." People take this to mean that it is not productive for developers to spend hours tuning the performance on one small component, only to find out that it is not invoked nearly so often, and the real limiting factor on performance was elsewhere. It is therefore considered unwise to start twisting knobs and tweaking settings until the performance results have come in.

However this does not mean that optimization *itself* is evil, and that you should ignore performance until right before you are ready to deploy your application. Unfortunately, a lot of organizations save their performance testing until the very end. This is when sometimes they get an unpleasant surprise. The biggest impact to an application's performance is often in the architecture and design decisions taken earlier in the development lifecycle; by the time the application is ready for deployment, it is much too late to do anything about the application architecture. No amount of tuning, or throwing hardware at the problem will significantly improve the performance of a badly designed application or algorithm.

A better approach, and one with less risk to the project, is to not save all performance testing until the very end. Agile development methodologies, such as XP and Scrum, focus on developing software in an incremental manner, with working software components at the end of each development cycle. This is a good time to initiate some performance testing, too. Sure, as the application evolves, the performance figures from one test cycle to another might bear no relation to each other; nor will the baselines have meaning across development cycles. But this approach will flag issues early, and prevent surprises at the end of the project.

Step 1: Set Up a Test Bed

Clearly, you should not do your performance testing against a production system as that would affect the end users. You should also not performance test against a development environment, as it usually would not have the same configuration as the production one.

Some of the considerations for a good test bed setup for performance testing are:

- **Test system should mirror the production system:** Performance testing should be done on a test bed that closely resembles the production environment (i.e., have similar hardware, operating system, and software versions). A test environment does not have the same security and uptime

requirements as a production one, so you can save on cost by not requiring all that you have on a production environment.

- **Test database should mirror the production database:** An application may perform well with a small number of records in the database, but could experience performance degradation a great deal as the number of records in the table(s) increases. The data in the test database should mirror that in production as much as possible, both in terms of the volume of data, as well as the interrelation of database records. However, you should be careful about taking a dump from the production database and uploading it into the test system. Often the production database contains information that should be kept confidential for privacy as well as legal reasons; it can include financial data, names and addresses of customers, and other sensitive information. Production data should be "cleaned" of all such information before you use it for testing.

Once the test bed is set up, the next step is to run tests and identify the performance baseline.

Step 2: Test Performance and Identify the Baseline

Before you start performance tuning in earnest, measure how the system is performing currently. The performance characteristics that you should be looking at include request latency/response time, throughput, and a measure of the Web site's scalability. These initial performance characteristics can be used as a means to compare future performance (i.e., they act as your performance *benchmark* or *baseline*).

Developing a good plan for performance testing is important. Some considerations while developing test plans are as follows:

- **Use automated performance test tools:** Use automated tools to load your Web application with traffic. The previous chapter covered the JMeter framework for performance testing. Use this, or any other tool — open source or commercial — to generate the request load.
- **Model expected Web site traffic:** The test traffic generated should mirror the expected traffic on the Web site as much as possible.
- **Test coverage:** Send test traffic to cover as many paths through your application as feasible, and send HTTP requests with different request parameters.
- **Model Web traffic spikes:** Web traffic will not always be uniform, and might have short term spikes or bursts. Generate test traffic of this nature, too, and measure the performance characteristics — or whether your site can handle such a load at all.
- **Test with the extremes:** Run your performance test in such a way that it tests the boundary conditions. For example, a page performing a search may work adequately when the search criteria returns a small result set but not so well when the search criteria returns significantly more data. Similarly, a Web page that performs well with low request volume can cause the server to fail under a higher request volume.
- **Test over longer periods:** Final tests should be over longer periods — typically days. The performance of your application and the JVM performance changes over time and can even improve if you are using HotSpot. Also, issues relating to memory leaks, temporary unavailability of databases, and other resource-related problems can sometimes be found only when running tests over longer periods.

It is important to collect data while running performance tests for baseline as well as all subsequent runs. Again, the previous chapter covered how you do this using the JMeter framework.

Step 3: Diagnose Performance Bottlenecks

Next, identify your performance bottlenecks. Performance issues can be diagnosed in a number of ways:

- Instrument your application with log messages, or profile it to determine where it spends most of its time.
- Use any performance tuning tools supplied by your database vendor to identify performance problems.
- Use the tools provided by your operating system to look at the CPU, memory, and I/O usage characteristics of your application. These can often give you a clue about what the root cause of the performance issue is.

Log messages can be extremely useful in situations where performance problems are difficult to duplicate. In some instances, a particular problem happens only when the application has been running for a while, or when a yet-to-be-discovered event happens — such as a specific pattern of input requests. Chapter 19 covers the logging implementations (log4j, JULI) that you can use in your Web application. Other than logs that the application writes, Tomcat itself writes log messages. Looking at these messages can also be instructive, especially if there are errors. Review these log files often.

Some of the useful Linux tools that you can use to examine the system resource usage include:

- **top:** Displays the top CPU-intensive processes.
- **vmstat:** Reports on virtual memory, as well as statistics on processes, and disk and CPU activity.
- **free:** A snapshot of the free and used memory on your system.
- **sysstat:** A family of tools available from `http://perso.orange.fr/sebastien.godard/index.html`. These include the `iostat` tool that reports on CPU and I/O statistics.

Windows comes with its own set of tools, such as the `Task Manager` and `PerfMon`.

Diagnosing Tomcat Performance Issues

Diagnosing issues with Tomcat can be very tricky. It gets easier when there is an indication about what is causing the problem in the Tomcat logs, such as:

- **Tomcat freezes or pauses with no requests being processed:** Tomcat could be doing a lot of garbage collection. Set the `-verbose:gc` flag for the JVM and observe from the log file if this is indeed the case. If so, tune the JVM parameters, such as `-Xms` and `-Xmx`. See the section "Tuning the JVM Parameters" later in the chapter.
- **OutOfMemory exceptions in the Tomcat logs:** This can occur for a number of reasons:
  - **A memory leak in your application:** Run a profiler to see if this is the case. The previous chapter listed some commercial as well as open source profiling tools.
  - **The maximum heap memory is less:** Use the `-Xmx` option to increase it, as explained in the section "Tuning the JVM Parameters" later in the chapter.

- **Too much memory has been allocated:** For example, you should not set your maximum heap size to be the same or very close to your system RAM. Your OS uses a certain percent of the RAM, and if the `-Xmx` value is set too high, Java can take this memory from your swap disk. This will slow down Tomcat significantly, and you might also see an `OutOfMemory` exception if you run out of swap.
- **Tomcat caches the JSP content generated:** If your generated JSP pages are very huge, the result may be `OutOfMemory` exceptions as the cached response fills up the heap. Set the `org.apache.jasper.runtime.JspFactoryImpl.USE_POOL` and `org.apache.jasper.runtime.BodyContentImpl.LIMIT_BUFFER` to `false` to fix this, as described in the section "Tuning JSP Tag Body Pooling" later in the chapter.
- **The JDK has built-in mechanisms to help debug pesky out-of-memory errors:** For instance, Sun's JDK has an option called `-XX:-HeapDumpOnOutOfMemoryError` that, as the name suggests, writes out the heap memory dump when there is an `OutOfMemory` exception. This `.hprof` file can then be read by profiling tools to help debug the problem.

- **The OutOfMemoryError:** PermGen space errors sometimes occur if you run multiple Web applications in single Tomcat instance, or if your application loads up a lot of classes. The JVM allocates a 64MB memory chunk for the permanent generation heap, which is the heap that holds objects such as classes and methods. When this space gets exceeded, you start getting the PermGen space errors. You would need to increase this setting: Use the `-XX:MaxPermSize` option in Sun's JDK to increase the permanent generation heap space. This is explained in more detail in the section "Tuning the JVM Parameters" later in the chapter.
- **In a setup with Web server (Apache/IIS) and Tomcat, incoming connections take too long or fail:** Look at Tomcat's logs for any `mod_jk` errors. This can indicate that all of Tomcat's AJP processors are in use and Tomcat either queued or rejected the connection. Increase the `maxThreads` setting, as described in the section "Tuning Tomcat Configuration" later in the chapter.

However, often things are not so straightforward, and you have to use your forensic skills to isolate the cause. Let's take an example scenario and work through some approaches for diagnosing the root cause. You start with the symptoms: In this example, the user sees poor performance while serving Web pages. The investigation for the root cause can proceed along the following lines:

- Check whether the poor performance occurs when serving both dynamic as well as static content. If it does, then it can rule out any database or Web application/Java issues. If not, then they are still a suspect area.
- The next step is to run `top` (assuming this is a Linux system) and see the processes running. You will probably see a bunch of processes for Tomcat and the JVM. Next, run `sar` and `vmstat`. This should tell you whether the processes are I/O bound or CPU bound.

  Processes generally fall into one of two categories: Either they are bound by the speed of the CPU (CPU bound) or the speed of the I/O (I/O bound). Database-heavy applications are typically I/O bound, and number-crunching applications are CPU bound. Scaling each of them is done differently — a CPU bound application typically requires a multi-CPU machine or clustering to scale. An I/O bound application may require a faster disk subsystem or perhaps an architecture change involving increased data caching.

- If they are I/O bound, then disk I/O (for example, database access, file system access) can be one issue.

- If they are CPU bound and the problem is in both static and dynamic pages, check the Web pages themselves. Perhaps server-side includes (SSI) have been enabled, which are leading to excessive CPU load.
- If the problem is only for dynamic Web pages, look at profiling the database interactions. P6Spy (`www.p6spy.com`) is a good open source profiler for database calls.
- If that doesn't show anything abnormal, look at the Web application itself. Run a Java profiler — the previous chapter lists a number of them — to isolate the cause of poor performance.

Tomcat Performance Tuning Tips

As mentioned earlier, performance tuning is a complex topic, and addressing everything that affects performance in one chapter is not possible. What this section will do, however, is first give you a feel of what can affect the performance of a Web application in general, and then address areas specific to Tomcat.

Broadly, factors that affect performance can be addressed at three stages: at design time, at development time, and at deployment time.

Decisions made at design time often have the greatest impact on performance. The design includes both the software as well as the system architecture. The software architecture is concerned with issues such as design of software modules, data structures, and so on. The system architecture, on the other hand, would also address issues such as the following:

- How will your software components be deployed?
- Is there load balancing to handle a large volume of requests?
- Are there clustering/failover capabilities to handle situations when a Web server or any other component of the Web site goes down?

Web application architecture is a complete subject in itself, and this chapter does not even begin to address these kinds of issues.

The development time issues relate to how the Java code for the Web application was designed and implemented. Again, there is a whole set of implementation best practices surrounding this area, such as:

- Do not create sessions for JSPs if they are not required.
- Do not store large objects in your session.
- Time out sessions quickly, and invalidate your sessions when you are done with them.
- Use the right scope for objects.
- Use connection pooling for improving performance.
- Cache static data.
- Use transfer objects to minimize calls to remote services.
- Minimize logging from Web applications, or use simple logging formats.

This book does not cover Java Web development. Refer to a book that does, such as *Beginning JavaServer Pages* (ISBN 0-7645-7485-X), for more details.

This chapter instead focuses on issues specific to deployment of Web applications in Tomcat. After a Web application is developed, it is deployed on a server. This rest of the book covers deployment time considerations that affect performance. Some of these measures include:

- Tuning the JVM parameters
- Pre-compiling your JSPs
- Tuning Tomcat's configuration, such as disabling the ability to check for JSP modifications
- Using a Web server, such as Apache or IIS, to serve static content

In addition to these, there are a host of other deployment time measures, such as tuning the parameters of the database that the Web application connects to, the operating system it runs on, and so on. Again, these are not covered in this chapter, and only Tomcat-related considerations are examined.

Tuning the JVM Parameters

The JVM has received quite a few "dials and knobs" in recent years that allow administrators to tweak its settings. The techniques described here are specific to IBM's and Sun's JVM implementations, and some of these are non-standard options. Refer to your JVM's documentation when in doubt!

Considering the Server VM

Surprisingly few system administrators realize that the JVM actually contains two different virtual machines inside the binary that's executed to start Java applications: the client VM and the server VM. Each of these two VMs is optimized according to the needs of client and server applications. The client VM's top priority is reducing startup time and minimizing the latency produced when the garbage collector reclaims memory. The server VM trades these priorities for an emphasis on greater scalability for server-type applications.

By default, Java uses the client VM. The server VM can be selected by passing the `-server` command-line option to the Java VM on startup. This can be done by setting the `JAVA_OPTS` variable in the Tomcat startup script. Administrators should experiment with this setting to increase their application's performance. Never assume that the server VM will be faster, however. Test the application with JMeter to see for yourself.

Optimizing Memory Allocation

The JVM will not take more memory from the operating system than it has been given permission to consume, and Tomcat will report an Out of Memory Exception when it runs out of memory. This limit is configurable via command-line switches to the JVM. These two JVM switches, shown in the following table, work in both Sun's and IBM's JVM.

Argument	Description
`-Xms<size>`	The initial heap size for the JVM
`-Xmx<size>`	The maximum heap size for the JVM

These, and other options, can be passed to the JVM running Tomcat by modifying the `<TOMCAT_HOME>/catalina.sh` (or `catalina.bat`) file, and adding the following line (as an example) in the beginning of the file.

On Linux:

```
JAVA_OPTS="$JAVA_OPTS -Xms512m -Xmx1024m"
```

On Windows:

```
SET JAVA_OPTS=%JAVA_OPTS% -Xms512m -Xmx1024m
```

In the previous example, the initial heap size is set to 512MB, and the maximum heap size to 1024MB — i.e., 1GB.

If these parameters are not explicitly set, the JVM uses its defaults. The defaults are dependent on the version of the JVM in use, but are generally too small for production use. The *heap* is the area of memory in which the JVM allocates new Java objects; the vast majority of the JVM's memory usage is its heap.

Initial heap size specifies the amount of operating system RAM to allocate to the Java heap at the time the JVM starts. Generally, this setting is not important. However, in a memory-intensive application under a heavy load, the initial heap size setting can have significant effects. If the JVM starts with a very small heap size and it is quickly pounded by many requests that require many object instantiations, the JVM must repeatedly increase its allocation until it reaches a sufficiently large size for the heap.

What are appropriate values for the initial and maximum heap size? Sun's JVM documentation recommends that for server applications, administrators should set the `-Xms` and `-Xmx` arguments to the same value. This improves the predictability of the garbage collector (GC), and helps in faster startups.

However, IBM's JVM documentation suggests that this might not always be a good idea because it delays the start of garbage collection until the heap is full. The first time the garbage collector runs, it is a very expensive operation. Also, the heap is more likely to be fragmented and require a heap compaction. Again this is a very expensive operation. Hence, IBM recommends that you start your application with the minimum heap size that it needs. This causes the garbage collector to be run often and, because the heap is small, efficiently.

The bottom line is that different JVMs may have different garbage collector algorithms implemented. Read your JVM's documentation and understand your application's memory profile before deciding on appropriate settings.

How do you understand your application's memory profile? Add the `-verbose:gc` option to your JVM parameters (again, via `JAVA_OPTS`) and restart your Web application. Run it with no load and look at the log files. You should see a log entry in this format:

```
[GC 32238K->4057K(520256K), 0.0281730 secs]
```

Here, `32238K` and `4057K` are the memory used by Java objects before and after garbage collection, and the value in parentheses — i.e., 520256K — is the total available space. These values with no load on the system give you an idea of how much heap space is required initially.

The maximum heap size specifies the upper limit of RAM that the JVM allocates to the heap. In a data-intensive application with long-lived objects, memory usage may quickly build up. If the memory required to run an application exceeds the maximum heap size configured, the JVM fails with a `java.lang.OutOfMemory` error. You should therefore set the maximum value possible for `-Xmx`.

Some of the additional options for IBM's JVM that allow you to control how the heap is managed are listed in the following table.

Argument	Description
`-Xmine<size>`	Set minimum size for heap expansion.
`-Xmaxe<size>`	Set maximum size for heap expansion.
`-Xminf<size>`	Set minimum percentage of heap free after GC.
`-Xmaxf<size>`	Set maximum percentage of heap free after GC.

An important option for Sun's JVM is the `-XX:MaxPermSize` option. This is used to specify the size permanent generation heap, which is the heap that holds objects such as classes and methods. As mentioned earlier in the chapter, if you start seeing errors like `OutOfMemoryError: PermGen space`, this is the option you should look to tweak. Sun's JVM reserves a space of 64MB, and it should be increased to a larger value, such as 128MB, if you start seeing these errors. Another workaround is to try another JVM, such as that from BEA or IBM.

For a complete list of the other performance tuning options in Sun's HotSpot JVM, see `http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp#PerformanceTuning`.

Choosing a Different Vendor's JVM

Sun isn't the only company producing Java virtual machines. Several other vendors (such as IBM and BEA) also produce JVMs compatible with the Sun Java specification, including the following:

- **IBM's JVM:** `www-128.ibm.com/developerworks/java/jdk/`
- **BEA's JRocket JVM:** `http://dev2dev.bea.com/jrockit`
- **Sun's JVM:** `http://java.sun.com/`

The performance characteristics of these JVMs are rarely "better" or "worse" than their competitors' products. Rather, they are typically better at *some* tasks, but *worse* at others. The best course of action is to test the application with a different JVM.

Precompiling JSPs

The first request for a JSP is handled differently from subsequent requests. When a browser requests a JSP page for the first time, the following tasks occur behind the scenes:

1. The Tomcat container translates the JSP to Java code in the form of a servlet.
2. This servlet is compiled into Java bytecode and this class file is kept in a file system location specific to the container.

3. The servlet class file is then loaded into memory.
4. The container calls an initialization method for the servlet.
5. The *service* method of this servlet is called. The service method handles the incoming request and generates the response that is eventually sent back to the user's browser.

Then, for every subsequent call to the JSP page, the following steps need to be performed:

1. The container first checks whether the JSP source file has changed since the last time it was compiled. This is typically done by comparing the timestamps on the files.
2. If there had been a change, the container would, as before, translate the JSP to a corresponding servlet Java code, and then compile it to a class file.
3. If nothing was changed in the JSP file, the container checks to see whether the servlet class corresponding to the JSP page has been loaded into memory.
4. If the servlet class is not in memory, the container loads and initializes it.
5. As before, the service method of this servlet is called. The service method handles the incoming request and generates the response that is eventually sent back to the user's browser.

This lifecycle of JSPs is summarized in Figure 21-1.

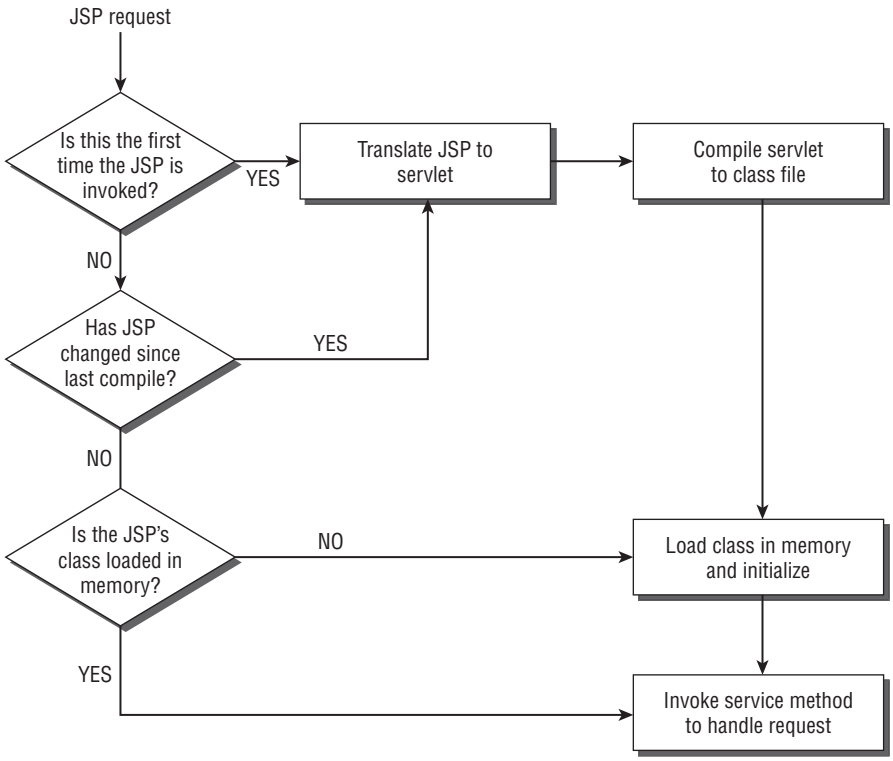

Figure 21-1: JSP lifecycle

The difference between the first time invocation (when the JSP is translated and then compiled into byte code) and subsequent invocations is quite evident; the first time the JSP is called, the response is noticeably slower than all other times.

This overhead can be avoided if the JSP is *precompiled*. Precompiling JSPs involves manually doing the tasks that the container does for first-time invocation of JSPs:

1. Translate JSPs to Java code.
2. Compile the Java code.
3. Copy the class files to a location where they can be loaded along with the rest of the Web application classes.

Appendix B has an example Ant script for precompiling JSPs. While precompiling JSPs offers a very small performance benefit — and that, too, only for the first invocation of the JSP — it can be coupled with turning the JSP development mode off, as explained later in the chapter. This disables the check Tomcat makes to see whether JSP pages are out-of-date for each incoming request.

Tuning Tomcat Configuration

Tomcat has several settings that can affect its performance. Adjusting these settings is often the absolute easiest way to achieve performance increases.

Adjusting Connector Settings

Tomcat's `server.xml` file contains a `Connector` element for each port on which Tomcat listens. The performance-related attributes shown in the following table can be modified on the `Connector` element. These setting are tunable for the default HTTP Connector, as well as the new high-performance Connectors such as the NIO or APR Connectors.

Parameter	Description
`maxThreads`	This is the maximum number of threads allowed. This defines the upper bound to the concurrency, as Tomcat will not create any more threads than this. If there are more than `maxThreads` requests, they will be queued until the number of threads decreases. Increasing `maxThreads` increases the capability of Tomcat to handle more connections concurrently. However, threads use up system resources. Thus, setting a very high value might degrade performance, and could even cause Tomcat to crash. It is better to deny some incoming connections, rather that affect the ones that are being currently serviced.
`maxSpareThreads`	This is the maximum number of idle threads allowed. Any excess idle threads are shut down by Tomcat. Setting this to a large value is not good for performance; the default (`50`) usually works for most Web sites with an average load. The value of `maxSpareThreads` should be greater than `minSpareThreads`, but less than `maxThreads`.

Table continued on following page

Parameter	Description
`minSpareThreads`	This is the minimum number of idle threads allowed. On Tomcat startup, this is also the number of threads created when the Connector is initialized. If the number of idle threads falls below `minSpareThreads`, Tomcat creates new threads. Setting this to a large value is not good for performance, as each thread uses up resources. The default (`4`) usually works for most Web sites with an average load. Typically, sites with "bursty" traffic would need higher values for `minSpareThreads`.
`tcpNoDelay`	When this attribute is set to `true`, it enables the `TCP_NO_DELAY` network socket option. This improves performance as it disables the Nagle algorithm, which is used to concatenate small buffer messages, which decreases the number of packets sent over the network. While this may result in better response time in a non-interactive network application because it enables greater throughput, it results in slower response times in interactive client-server environments (such as a Web browser interacting with the Web server).
`maxKeepAliveRequest`	This attribute controls the "keep-alive" behavior of HTTP requests, enabling persistent connections (that is, multiple requests to be sent over the same HTTP connection). It specifies the maximum number of requests that can be pipelined until the connection is closed by the server. The default value of `maxKeepAliveRequest` is `100`, and setting it to `1` disables HTTP keep-alive behavior and pipelining.
`socketBuffer`	This specifies the size, in bytes, of the buffer to be used for socket output buffering. Use of a socket buffer helps to improve performance. By default, a buffer of size 9,000 bytes is used, and setting `socketBuffer` to `-1` turns buffering off.
`enableLookups`	Setting `enableLookup` to `false` disables DNS lookups for the `request.getRemoteHost()` API calls. This improves performance by decreasing the time required for the lookup.

The New High-Performance Connectors

In addition to the default HTTP Connector, Tomcat ships with two high-performance Connectors — the APR and the NIO Connectors. These Connectors have the potential to deliver far higher performance than the default HTTP Connector, with the caveat that they are relatively new implementations and have (at the time of this writing) stability issues.

The following table compares the behavior and capabilities of these three Connectors.

	Default HTTP Connector	NIO Connector	APR Connector
Support for polling	No	Yes	Yes

	Default HTTP Connector	NIO Connector	APR Connector
Maximum number of connections	Defined by setting `maxThreads` — see the section "Adjusting Connector Settings" earlier in the chapter.	Defined by the polling size, which is bound by system limits.	Defined by the polling size, which is bound by system limits.
Blocking read/write	Yes — both reads and writes block.	HTTP Request reads don't block, but writes do.	Yes — both reads and writes block.
SSL Implementation	Java SSL	Java SSL	Open SSL (`www.openssl.org`)
SSL Handshaking	Blocking	Non-blocking	Blocking

APR Connectors

Apache Portable Runtime (APR) is a platform-specific binary library that Tomcat 6 can use to boost its Web server performance. APR was originally developed as a part of the Apache 2 Web server, but is now used in many other projects. APR opens up the use of low-level, performant API calls such as `sendfile()` and `epoll()` in Tomcat, as well as allowing for the use of a non-Java SSL implementation (OpenSSL). Use of this library in Tomcat has the potential to deliver far higher performance — both in a standalone configuration, as well as when integrated with "native" Web servers such as Apache.

Java purists sometimes have issues with using APR because it is implemented in C. However, APR been compiled on almost all server platforms, so portability of APR is not an issue.

Using APR involves two steps:

- Install APR. Chapter 3 explains how to install APR. On Linux/Unix systems, you would also need to add the APR JNI wrapper to your `LD_LIBRARY_PATH`.
- Configure the Connector in `server.xml` to use the APR connector.

Chapter 3 explains how to install APR, and Chapter 10 explains how to configure the APR support in Tomcat's Connectors, as well as how to tune its configurable parameters for improved performance.

Tuning APR Connectors

As mentioned earlier, the APR Connector can be used both in a standalone configuration, as well as with a Web server such as Apache via the AJP Connector.

One of the major performance features of APR in the Tomcat standalone configuration (i.e., Tomcat working as a Web server, too) is the use of the kernel level `sendfile()` system call to handle large static files. This call is optimized for sending large static files through a socket, and instead of repeatedly copying the file data to higher level buffers, such as those maintained in byte arrays within the Java VM, the kernel mode API takes care of sending the file directly from the file system's buffer cache. Although this `sendfile()` operation is performed synchronously by the kernel, it is asynchronous with respect to the Java VM. This theoretically enables the Tomcat server to perform other work while the file is being sent by the lower level

call. This feature cuts down on CPU time spent on data copying as well as minimizes the context switches between the Java VM and kernel mode operations during the sending of very large static files.

On systems without the kernel mode `sendfile()` system call, the Tomcat connector gracefully falls back on the Java-based buffer IO to send large static files. In addition, the `sendfile()` operation does not take effect when SSL is used with the connector.

The configurable parameters for tuning the `sendfile()` behavior are discussed in the following table.

Parameter	Description
`useSendfile`	Use kernel level `sendfile()` API for certain static files. The default value is `true`.
`sendfileSize`	This is used to control the maximum number of sockets used by the poller for `sendfile()`–based operations. Default is `1024`. On platforms where the `sendfile()` call returns asynchronously, the poller is not used and the maximum number of concurrent static file transfers may not correspond to this value.

A sample APR Connector configuration is listed here. The `useSendFile` attribute need not be specified as it is true by default.

```
<Connector port="8080"
           protocol="org.apache.coyote.http11.Http11AprProtocol"
           maxThreads="150"
           connectionTimeout="20000"
           redirectPort="8443"/>
```

The APR Connector can also be used with AJP to connect to a Web server such as Apache. In this configuration, it cannot use `sendfile()` — nor does it need to — because static files are handled by Apache. Instead, the optimization provided by APR is the use of a socket poller for keep-alive, increasing scalability of the server. As AJP is designed around a pool of persistent connections, this reduces the amount of processing threads needed by Tomcat. The keep-alive poller is an APR component responsible for maintaining keep-alive connections. The number of kernel modes to the Java VM context switches is reduced when a native code component is used to keep track of keep-alive connections.

The configurable parameters for this keep-alive poller are listed in the following table.

Parameter	Description
`firstReadTimeout`	This value, in milliseconds, controls the timeout set on a connection's first read call. If this timeout is reached, the connection is handed off to the keep-alive poller. Note that if this timeout value is never reached, the keep-alive poller does not get involved with the connection. If you want to use the keep-alive poller every time, you can set the value to `0, -1`. Both hand off connections to the poller on every read; however, `0` tells the poller to use very short timeout and `-1` indicates the use of the connection's configured socket timeout value. Default value is `-1`. Note that `-2` can also be used to request that the runtime bypass the use of the poller as much as possible.

Parameter	Description
`pollTime`	Adjusts the timing between poll calls. The more frequent these calls are made, the faster the managed sockets will respond to keep-alive termination. However, this will also result in increased CPU load as a result of the more frequent polling. The default value specifies the number of poll calls made per second; the default is 2000 (polling 5ms apart). This means that in the worse case, a socket destined for keep-alive termination may not be detected until 5 ms after its targeted disconnection time.
`pollerSize`	This is used to control the maximum number of sockets available to the keep-alive poller. Default is 8192.

NIO Connectors

NIO, or New I/O, was introduced in JDK 1.4. It took a different approach from the standard "stream-oriented" Java I/O implementation. In particular, it introduced the following new features:

- A multiplexed, non-blocking I/O facility
- Support for data buffers

In addition to this departure from standard Java I/O, NIO has other minor new features, too, such as a pattern-matching implementation, and a file implementation that allows for file locks and memory mapping.

> *At the time of this writing, the stable Tomcat 6 release does not have a production quality NIO Connector — the version shipped with Tomcat 6.0.10 has known memory leaks. Hence, check the status/stability of the NIO Connector before planning to use it in a production deployment. Finally, a caveat common to all new open source implementations — the configuration options might change, so refer to the NIO Connector documentation when in doubt.*

Here, multiplexing refers to the sending of multiple signals or streams over the same transport. In NIO, this maps to multiple open socket calls to an NIO *selector*. This NIO selector is able to handle these multiple open sockets, and hence multiple requests with a single thread. This is very significant, and the central reason why NIO scales much better than standard Java IO.

The new NIO Connector is a Tomcat Connector implemented using the NIO API, and designed for highly scalable servers. A major issue with scaling Tomcat is threads — each incoming request to the standard Tomcat Connector gets a thread assigned to it, and this thread lives on until the request is served. As the number of concurrent requests increases, the CPU becomes overburdened by context switching between the threads. And finally, there is an upper bound on the number of threads that can be supported, and hence on the number of concurrent requests.

NIO solves this problem by allocating a fixed pool of threads, and serving incoming connections from this pool. Performance benchmarking done with NIO and the standard IO-based Connectors have shown that under moderate load both perform about the same. However, as the number of concurrent incoming connections increases, Tomcat's performance degrades very rapidly. As mentioned earlier, this

is primarily because of the cost of context switching between so many threads. The NIO Connector, on the other hand, has performance degrade in a linear fashion.

So is it is always better to use the NIO Connector?

Well, having the Tomcat HTTP Connector allocate one thread for each incoming request is a good thing when the request needs to perform a lot of operations on the server because there is a dedicated thread. This request thread blocks others until it has a response for the client. Hence, this often leads to a lower latency for clients.

However, allocating a thread for every request soon hits system limits when the number of concurrent requests goes up. NIO's use of thread pools leads to better scalability here, although sometimes at the cost of latency. If latency becomes an issue with NIO, you can switch to the default Tomcat HTTP Connector, and scale your Web site by "scaling up" (i.e., adding a beefier server), or "scaling across" (i.e., Clustering, as shown in Chapter 17).

The bottom line, as always, is to benchmark your application with both Connectors and with varying request loads to decide which works best for you.

Tuning NIO Connectors

The NIO Connector, unlike the default HTTP Connector, exposes low-level, socket-level configuration parameters to the administrator. While the list of options seems long and daunting, the default settings are good enough to work with.

These NIO Connector configuration properties include those listed in the table that follows.

Parameter	Description
`acceptorThreadCount`	The number of threads to be used to accept connections. `acceptorThreadCount` defaults to 1 thread. Increase this value on a multi-CPU machine.
`pollerThreadCount`	The number of threads to be used to run for the polling events. The default value is 1.
`selectorTimeout`	The time in milliseconds to time out on a `select()` system call for the poller. This value is important because the cleanup for connections is done on the same thread, so the documentation suggests that you don't set this value to an extremely high one.
`useComet`	Allow comet servlets or not. This parameter defaults to `true`. Comet support allows a servlet to process IO asynchronously. This means it can receive events when data is available for reading on the connection and also for writing data back on connections asynchronously.
`socket.directBuffer`	Use direct ByteBuffers (`true`) or java mapped ByteBuffers (`false`). This attribute defaults to `false` (i.e., Java mapped ByteBuffers).When you are using direct buffers, make sure you allocate the appropriate amount of memory for the direct memory space — for example, 256MB. On Sun's JDK that can be done via the `-XX:MaxDirectMemorySize` command-line parameter.

Parameter	Description
`socket.rxBufSize`	The socket receives the buffer (`SO_RCVBUF`) size in bytes. The default value is 25188.
`socket.txBufSize`	The socket sends the buffer (`SO_SNDBUF`) size in bytes. The default value is 43800.
`socket.appReadBufSize`	Each connection that is opened up in Tomcat gets associated with a read and a write ByteBuffer. This attribute controls the size of these buffers. By default, this read buffer is sized at 8192 bytes. For lower concurrency, you can increase this to buffer more data. For an extreme amount of keep-alive connections, decrease this number or increase your heap size.
`socket .appWriteBufSize`	Each connection that is opened up in Tomcat get associated with a read and a write ByteBuffer. This attribute controls the size of these buffers. By default, this write buffer is sized at 8192 bytes. For low concurrency you can increase this to buffer more response data. For an extreme amount of keep-alive connections, decrease this number or increase your heap size.
`socket.bufferPool`	The Nio Connector uses a class called `NioChannel` that holds elements linked to a socket. To reduce garbage collection, the Nio Connector caches these channel objects. This value specifies the size of this cache. The default value is 500, and represents that the cache will hold 500 `NioChannel` objects. Other values are –1. unlimited cache, and 0, no cache.
`socket.bufferPoolSize`	The NioChannel pool can also be size-based, not used-object–based. The NIO documentation states that size is calculated as follows: — NioChannel buffer size = read buffer size + write buffer size — SecureNioChannel buffer size = application read buffer size + application write buffer size + network read buffer size + network write buffer size — The value is in bytes, the default value is 1024*1024*100 (100MB)
`socket.keyCache`	Tomcat caches `KeyAttachment` objects to reduce garbage collection. The integer value specifies how many objects to keep in the cache at most. The default is 500. Other values are –1. unlimited cache, and 0, no cache.
`socket.eventCache`	Tomcat caches `PollerEvent` objects to reduce garbage collection. The integer value specifies how many objects to keep in the cache at most. The default is 500. Other values are –1. unlimited cache, and 0, no cache.
`socket.tcpNoDelay`	Same as the standard setting `tcpNoDelay`. The default value is `false`.

Table continued on following page

Parameter	Description
socket.soKeepAlive	Boolean value for the socket's keep-alive setting (SO_KEEPALIVE). The default is false.
socket.ooBInline	Boolean value for the socket OOBINLINE setting. Default value is true.OOB, or Out Of Band data, is data sent asynchronously. If OOBINLINE is enabled, the receiving process will be interrupted when out-of-band data arrives.
socket.soReuseAddress	Boolean value for the sockets reuse address option (SO_REUSEADDR). Default value is true.
socket.soLingerOn	Boolean value for the sockets so linger option (SO_LINGER). The default value is true. This option is paired with the soLingerTime value.
socket.soLingerTime	Value in seconds for the sockets so linger option (SO_LINGER). The default value is 25 seconds. This option is paired with the soLinger value.
socket.soTimeout	Value in milliseconds for the sockets read timeout (SO_TIMEOUT). The default value is 5000 milliseconds.
socket.soTrafficClass	Value between 0 and 255 for the traffic class on the socket, 0x04 | 0x08 | 0x010.
socket .performanceConnectionTime	A value that expresses the relative importance of a short connection time. Defaults to 1. The absolute value of this setting is not important, just the relation with socket.performanceLatency and socket.performanceBandwidth. If you wish to give preference to high bandwidth over low latency, and low latency over short connection time, then set the values as performanceConnectionTime=0, performanceLatency=1, and performanceBandwidth=2.
socket .performanceLatency	A value that expresses the relative importance of low latency. The default is 1. See performanceConnectionTime.
socket .performanceBandwidth	A value that expresses the relative importance of high bandwidth. The default is 1. See performanceConnectionTime.
selectorPool .maxSelectors	The max selectors to be used in the pool, to reduce selector contention. Use this option when the command-line o.a.t.u.net.NioSelectorShared value is set to false. The default value is 200.
selectorPool .maxSpareSelectors	The max spare selectors to be used in the pool, to reduce selector contention. When a selector is returned to the pool, the system can decide to keep it or let it be garbage collected. Use this option when the command-line option org.apache.tomcat.util.net .NioSelectorShared value is set to false. The default value is -1 (i.e., unlimited).

A sample `server.xml` configuration for an NIO Connector is shown here:

```
<Connector port="8080"
           protocol="org.apache.coyote.http11.Http11NioProtocol"
           maxThreads="150"
           connectionTimeout="20000"
           acceptorThreadCount="2"
           redirectPort="8443"
           socket.directBuffer="false"/>
```

As you can see from this, the default Connector configurable parameters, such as `maxThreads`, are still valid for NIO Connectors.

In addition to these options, there are also command-line options as listed. As before, add a `CATALINA_OPTS` parameter setting in the Tomcat startup script to set this value.

- `org.apache.tomcat.util.net.NioSelectorShared`: Set this value to `false` if you wish to use a selector for each thread. If you do set it to `false`, you can control the size of the pool of selectors by using the `selectorPool.maxSelectors` attribute. `NioSelectorShared` defaults to `true`.

Adjusting JSP Settings

A number of JSP configuration settings affect performance. These are listed in the following table.

Parameter	Description
`development`	Setting this to `false` disables Tomcat from checking the JSP pages for modification.
`reloading`	If `reloading` is set to `true`, background compiles are enabled. The background compile option causes the Tomcat container to check and compile JSPs after a period of time.
`checkInterval`	If `reloading` is enabled, background compiles are enabled, too. The `checkInterval` decides how frequently the compiles are triggered. This defaults to 300 seconds.
`modificationTestInterval`	This sets the frequency for checking the modification of JSP files. This defaults to `4` seconds, and setting it to `0` causes the JSP to be checked on every access. If `development` and `reloading` are set to `true` for any reason (such as dynamic generation of JSPs), setting this to a high value improves performance a lot.
`genStringAsCharArray`	To generate slightly more efficient char arrays, set this to `true`.
`enablePooling`	The `enablePooling` attribute specifies if pooling of Tag library classes is to be enabled (`true`) or not (`false`). This parameter defaults to `true`.
`trimSpace`	To remove useless white space bytes from the response, set this to `true`.

Turning Off JSP Reload and JSP Development Mode

The JSP lifecycle in Figure 21-1 showed how each time there is a request for a JSP, the container first checks to see whether it has been modified. This feature is useful for a development environment, as it allows you to make changes to JSPs and see the effect without redeploying the Web application.

However, in a production environment, this can affect performance of JSP pages. Often in such environments, this JSP modification time check is disabled.

The following is an excerpt from the global Web application configuration file (`<Tomcat Home>/conf/web.xml`) that shows how this parameter can be tuned for the Jasper servlet. The Jasper servlet, as mentioned earlier, invokes the JSP page compiler.

The two attributes that control the JSP compilation behavior in Tomcat are `development` and `reloading`. If you don't want the JSP pages to be checked for modification each time there is a request, you should set `development` to `false`.

In the following settings, the `reloading` attribute is also set to `false`. Had `reloading` been `true`, background compiles would have been enabled. The background compile option causes the container to check and compile JSPs after a period of time. Another configurable parameter, called `checkInterval` decides how frequently the compiles are triggered.

In the current Tomcat default setting, both `development` and `reloading` are set to `true`, and `checkInterval` is set to `300` seconds.

```
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>fork</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>development</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>reloading</param-name>
        <param-value>false</param-value>
    </init-param>
    ...
</servlet>
```

The modification time check feature enables you to use JSPs as a scripting tool in development environments. In these environments, you can make changes to JSP files and immediately see the effect. When you are finally ready for production, you can simply turn this off and precompile the code for better performance.

Turning On Custom Tags Pooling

Custom JSP tags are a recommend "best practice" for keeping Java code out of JSPs, and using a tag library (JSTL or custom) makes for more maintainable code. However, a clean and maintainable design

does not come without some costs, and the excessive use of customs tags affects performance. When there is a design tradeoff between a maintainable design and squeezing out a little bit more performance, maintainable design should be chosen; it pays off in the long run in terms of programmer productivity.

Jasper (the JSP to Java translator for Tomcat) addresses the issue of tag performance to some extent by supporting custom tag pooling. The Java objects instantiated for the custom tags are now pooled and reused and this boosts the performance of JSP pages.

The following is an excerpt from the global Web application deployment descriptor (`<Tomcat Installation Directory>/conf/web.xml`) that has defaults for all the Web applications. The settings are for the `JspServlet`, which handles all requests for JSP pages. The `enablePooling` attribute specifies whether pooling of tag library classes is to be enabled (`true`) or not (`false`). If this setting is missing, it defaults to `true`.

```
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>fork</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>enablePooling</param-name>
        <param-value>true</param-value>
    </init-param>
     ...
</servlet>
```

Tuning JSP Tag Body Pooling

Additional JSP settings can be configured via system properties:

- `org.apache.jasper.runtime.JspFactoryImpl.USE_POOL`: This property can be set to either `true` or `false`, and affects whether the JSP tag bodies are pooled or not.
- `org.apache.jasper.runtime.JspFactoryImpl.POOL_SIZE`: This specifies the pool size for the tag body pooling (i.e., the number of JSP page contexts to be pooled). `POOL_SIZE` defaults to 8.
- `org.apache.jasper.runtime.BodyContentImpl.LIMIT_BUFFER`: Specifies whether to do buffering of characters while writing out JSP tag bodies (`true`) or not (`false`).

These values can be set by editing the Tomcat startup script (`catalina.bat`, `catalina.sh`) and adding the settings to the `CATALINA_OPTS` parameter. For example, on Linux:

```
CATALINA_OPTS="-Dorg.apache.jasper.runtime.JspFactoryImpl.USE_POOL=true"
```

The `USE_POOL` parameter decides whether to use object pools for the bodies of the JSP tags. This helps performance because it makes JSP processing significantly faster. For this reason, the `USE_POOL` parameter setting defaults to `true`.

Is it always a good idea to pool the JSP tag bodies? In most cases it is. However, if your JSP pages are large, and have tags with huge bodies, it can eat up a lot of memory — eventually even running out of memory when the number of concurrent sessions goes up. If you profile your application and see a lot of memory held by `BodyContentImpl`, then you have identified your problem. Turn off JSP tag body pooling by adding the following line to your `catalina.sh` script:

```
CATALINA_OPTS="-Dorg.apache.jasper.runtime.JspFactoryImpl.USE_POOL=false"
```

This enables you to scale your application. However, note that there is a tradeoff here, as this option was set to `true` by default to prevent a lot of garbage collection, and hence make JSP handing faster.

If you are running into memory issues with large JSPs, you should also try turning off buffering of characters while writing out JSP tag bodies by setting the `LIMIT_BUFFER` parameter to `false`.

Turning Off Web Application Auto-Deploy and Reloading

By default, Tomcat monitors the `<TOMCAT_HOME>/webapps` directory for new Web applications, and as soon as a WAR file is copied there, it gets automatically deployed. While this auto-deploy feature is great for development environments, it has a performance impact and should be disabled in production.

To turn off auto-deploy in Web applications, edit the `server.xml` file and set the `autoDeploy` attribute in the appropriate `Host` to `false`, as shown here:

```
<Host name="localhost" appBase="webapps" unpackWARs="true"
      autoDeploy="false"
      xmlValidation="false" xmlNamespaceAware="false">
```

Tomcat can also monitor classes under each Web application's `/WEB-INF/classes/` and `/WEB-INF/lib` for changes, and automatically reload the Web application if a change is detected. Again, this Web application "reloading" functionality has runtime overhead and is not recommended for use in production environments. That's why even the default setting for this attribute is `false`. This option (`reloadable`) can be set on the Web application's Context in the `web.xml` file.

Using Web Servers for Static Content, When Appropriate

Web sites contain both static and dynamic content. The static content includes HTML pages, images, and style sheets (.css files), whereas the dynamic content includes JSPs and servlets.

All the examples in this book have been shown as running within a servlet container (Tomcat). While this is fine for development time, often for production deployments, a Web server is used in addition to a servlet container or a J2EE application server. In such deployments, the Web server, for example Apache or IIS, serves up the static content to Web browsers. When there is a request for JSP or a servlet, a special configuration at the Web server end tells it how to send the request to the Servlet container. The Servlet container then processes the request, and returns the result to the Web server, and finally to the user's browser.

Figure 21-2 shows a typical deployment.

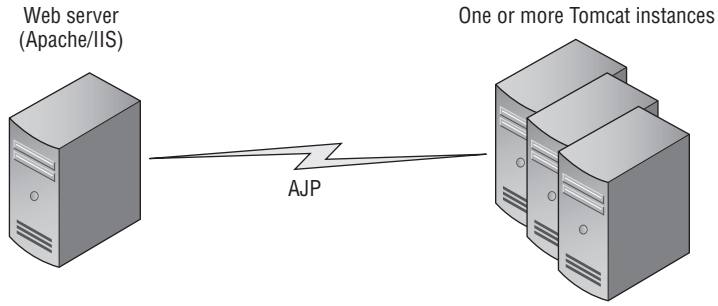

Figure 21-2: Using a Web server in addition to Tomcat

Do you always need to have such a deployment? Some reasons for using a Web server to front Tomcat include the following:

- **Stability:** Such a configuration is more stable than a standalone one involving just a Servlet container; Web servers such as Apache are typically a lot more robust. If the Servlet container crashes, for instance, the rest of the Web site that contains static content is still available to users.
- **Clustering and load balancing:** The JK Connector has a good support for clustered Tomcat configurations, as well as load balancing. Chapter 17 walks you through this in greater detail. However, this is not the only way to do clustering and/or load balancing.
- **Legacy support:** Web sites sometimes have legacy modules implement in other languages such as Perl, Python, and ASP. Using a Web server frontend such as Apache or IIS enables you to continue using their functionality.

Performance *can* be one of the reasons for selecting a Web server frontend, but this depends on the nature of your Web application. If your application has mainly dynamic content — i.e., servlets and JSPs — you will not see a significant performance difference by using Apache or IIS as a Web front to Tomcat. In fact, performance might decrease because an extra layer needs to be traversed by incoming requests.

However, you may get improved performance if your Web application has the following characteristics:

- It has a lot of static content, such as images, multimedia content, HTML pages, and so on.
- It uses SSL extensively.

In any case, the way to decide if one configuration is better than the other performance-wise is to benchmark the performance for your Web application. Build test beds with both configurations (standalone Tomcat, Tomcat with Apache/IIS), deploy your application, and run tests. Test with the new high-performance Connectors such as APR or NIO. The proof of the performance of either configuration — standalone or with a Web server frontend — is in testing it.

The bottom line: Let the Web server do what it does best — i.e., serve static content; and let Tomcat do what it does best — serve up dynamic servlet and JSP content. If your Web application is mainly servlets and JSPs, you might be fine with just a standalone Tomcat deployment.

Chapter 10 explains how to use Tomcat in a standalone deployment, and Chapters 11 and 12 show how to use Apache and IIS as Web server frontends to Tomcat.

Summary

This chapter focused on introducing best practices for performance testing, as well as summarizing all the performance tuning tips and settings referred to elsewhere in the book. At the end of this chapter, you should:

- Understand the performance tuning process: writing a test plan, setting up a test bed, running tests, comparing against the baseline, diagnosing bottlenecks, and fixing problems in a controlled manner.
- Appreciate that performance testing is not something to be done just before the production launch of a Web site. Often, the decisions that affect performance are made a lot earlier in the development lifecycle, and it is hard to compensate for bad design or implementation.
- Be aware of the mechanisms available for diagnosing performance bottlenecks. These include log files; system resource monitoring tools such as `top`, `sar`, `iostat`, and `vmstat` on Linux and the `Task Manager` and `PerfMon` on Windows; and profiling tools — both for Java applications (JProbe, OptimizeIt, Eclipse TPTP), as well as for SQL calls (P6Spy).
- Know the Tomcat configuration parameters that can be tweaked for better performance.

Performance tuning is a very broad and complex area, and difficult to cover completely in one chapter. The aim of this chapter was to expose you to most of the common areas of concern for performance tuning.

Tomcat and IDEs

Java has a rich set of Integrated Development Environments (IDEs), both open source as well as commercial. These IDEs enable application developers to develop, compile, debug, test, and deploy Web applications — all from within the IDE itself.

This appendix outlines the support for Tomcat in the following IDEs:

- Eclipse
- NetBeans

At the end of this appendix, you will be able to use either of these IDEs to debug Web applications deployed to a Tomcat server — both a local server as well as a remote one.

This appendix covers the two widely used open source Java IDEs, but does not cover the commercial ones such as Intellij IDEA (`www.intellij.com`) and JBuilder (`www.borland.com/jbuilder`). However, some of the steps are similar across IDEs, such as those for remote Web application debugging in Tomcat. The appendix also does not cover installation of either Eclipse or NetBeans, or how to set up a project in these IDEs.

Eclipse

Eclipse is a popular, open-source IDE developed by IBM. It was originally designed to be a Java IDE, but through extensions it now supports other languages such as C/C++ and Ruby, even COBOL. It also has extensions ("plugins") for a host of tools and applications, such as Apache Ant, JUnit, and many more.

Eclipse can be downloaded from the `eclipse.org` Web site. The version of Eclipse used for this appendix is 3.2.2. You also need to install a JDK, and have your JAVA_HOME set to the JDK install directory. Chapter 3 shows you how to do this.

You can go about deploying and debugging Web applications in Tomcat through Eclipse in a number of ways:

- Deploy using an Ant build script to either a local or remote Tomcat instance; and use the remote debugging feature of the JVM to debug the Web application from Eclipse.
- Use the Sysdeo Tomcat plugin to deploy to a local Tomcat instance. This allows you to start and stop Tomcat, and debug the Web application, too.
- Use the Web Tools Platform (WTP) to deploy to a local or remote Tomcat instance. This, too, allows you to start and stop Tomcat, and debug the Web application.

Although the last two points are very convenient and productive for development environments, there is something to be said about using the Ant build script (either from within the Eclipse IDE or from the command line) to manage build and deployment. This allows you to standardize on build procedures, as a common Ant script can be used by all developers. Appendix B also covers how to develop reusable build scripts that can be used for different environments (development, QA, production), as well as different operating systems.

Debugging a Remote Web Application in Eclipse

Starting with the JDK 1.3, the Java Virtual Machine (JVM) gained the capability to pass debugging information to external applications and receive debugging commands from them. The data can be transferred via either a network socket or shared memory on a local machine. Tomcat fully supports remote debugging. IDEs that support remote debugging, such as Eclipse, can thus attach to a Tomcat instance remotely to step through Java code. Eclipse, however, supports only the network socket mechanism for debugging.

To enable remote debugging, you need to perform two steps: First specify the `JPDA_TRANSPORT` and `JPDA_ADDRESS` to the Tomcat JVM, and second, configure remote debugging parameters in Eclipse. JPDA, or Java Protocol Debugger Architecture, is the mechanism provided for debugging Java programs, both locally as well as remotely. The `JPDA_TRANSPORT` parameter specifies the transport to use, such as socket or shared memory, and `JPDA_ADDRESS` specifies the transport-specific address, such as a port number in case of network sockets.

To specify the JPDA parameters, modify the `catalina.bat` or `catalina.sh` script under `<TOMCAT_HOME>/bin`, and add the following environment variables to the start of the script.

For Linux, add the following:

```
JPDA_TRANSPORT=dt_socket
JPDA_ADDRESS=9000
```

For Windows, add the following:

```
SET JPDA_TRANSPORT=dt_socket
SET JPDA_ADDRESS=9000
```

The port number (specified in `JPDA_ADDRESS`) can be arbitrary, as long as it is not in use by any other process.

To launch Tomcat in debug mode, use one of the following commands.

For Windows, use this command:

```
%CATALINA_HOME%\bin\catalina jpda start
```

For Linux, use this command:

```
$CATALINA_HOME/bin/catalina jpda start
```

What this does is pass the following options to the Tomcat JVM. You can even set the `JAVA_OPTS` directly in the `catalina.bat/catalina.sh` script; this enables you to have remote debugging when the Tomcat startup is not from the command line (for example, as a service).

```
JAVA_OPTS="-Xdebug -Xrunjdwp:transport=dt_socket,address=9000,server=y,suspend=n"
```

Restart Tomcat after making these changes. Look at Tomcat startup log messages, and find a log message such as `Listening for transport dt_socket at address: 9000`. This indicates that the changes you made have been picked up by Tomcat.

Next, to configure Eclipse, perform the following steps:

1. Configure the Web application's source code as an Eclipse project. Set a break point at a location in the Web application code that you wish to debug.
2. Select the Run pull-down menu, and then select the Debug option. You should now see a dialog box, as shown in Figure A-1.

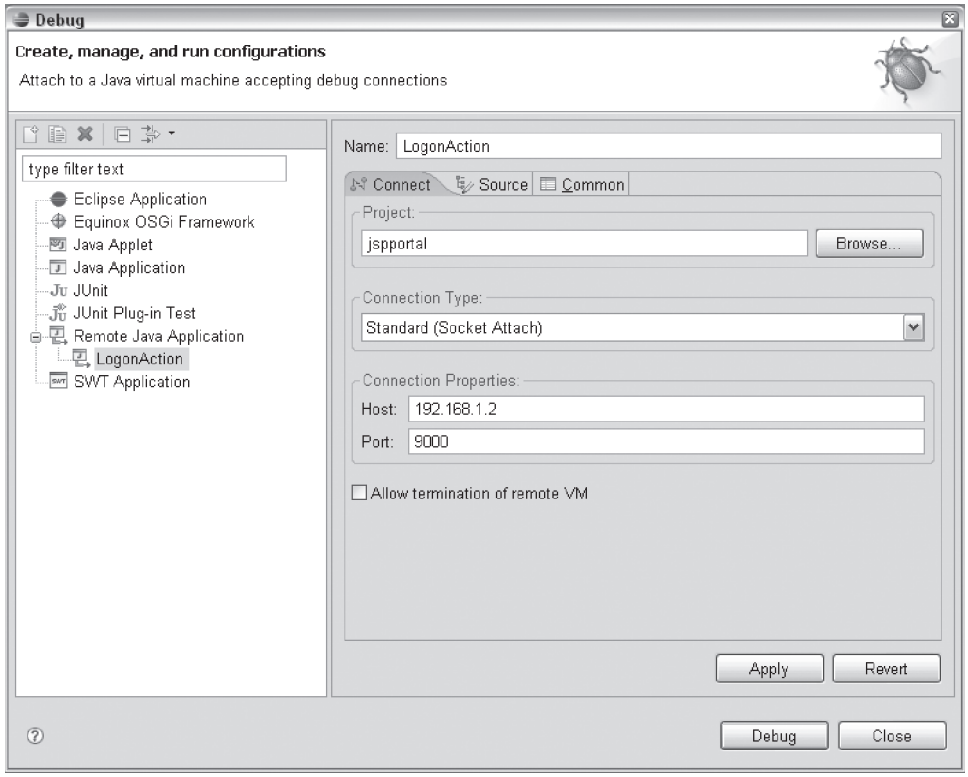

Figure A-1: Configuring the host and port in Eclipse for remote debugging

3. Double-click on the Remote Java Application node.
4. Select the appropriate project in the Project field.
5. Change the Port field to that specified in the `JPDA_ADDRESS` variable (here `9000`).
6. Specify the IP address or host name of the machine in the Host field. This can be `localhost` if Tomcat is running on the same machine as Eclipse.
7. Click the Debug button.

Eclipse should indicate that it is connecting to the remote Java process. You might sometimes get an error message, such as `Failed to connect to remote VM. Connection refused`. Typically, this error results because of problems with network permissions in connecting to the specified host/port or multiple debugging instances trying to connect to the same Java process.

If everything goes well, you can now run the Web application in a browser. Once the code in the Web application reaches the location where you have your break point set, Eclipse gives you an alert and switches to the debug mode, as shown in Figure A-2.

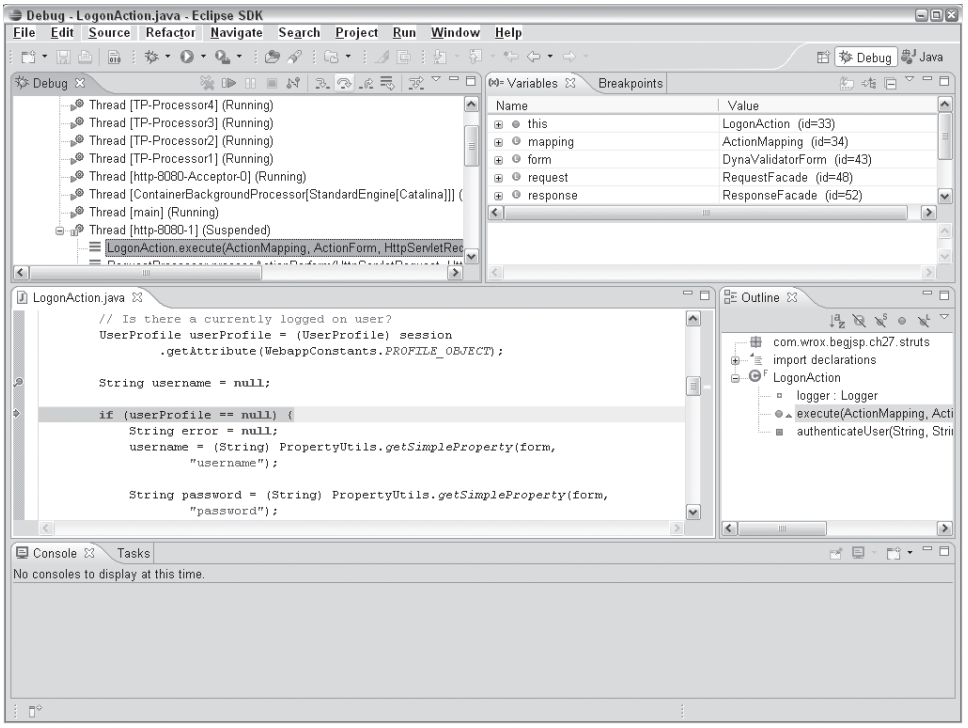

Figure A-2: Debugging a Web application in a remote Tomcat instance

Deploying and Debugging Local Web Applications Using the Sysdeo Tomcat Plugin

Originally, Eclipse did not include Tomcat support out of the box so a lot of third-party plugins filled the void. The most popular of these is the Sysdeo plugin, which can be downloaded from `sysdeo.com/eclipse/tomcatplugin`. The version of Sysdeo used in this example is 3.2 beta3, which supports Tomcat 6. Sysdeo does not include Tomcat within it; instead, it requires that you install Tomcat separately. This allows you to use a Tomcat version of your choice.

Installing Sysdeo is straightforward:

1. Download the plugin file, `tomcatPluginVxxx.zip`, where *xxx* is the version number. For example, the version used in this chapter was `tomcatPluginV32beta3.zip`. Unzip it into the `<ECLIPSE_HOME>\plugins` directory. You should now see a directory named `<ECLIPSE_HOME>\plugins\com.sysdeo.eclipse.tomcat_3.2.0.beta3` (depending on the Sysdeo version number).
2. Launch Eclipse using the command-line option `-clean`. This option enables Eclipse to detect a new plugin, and is required only once.
3. You should now see the Tomcat icons and menu options in Eclipse that allow you to start, stop, and restart Tomcat (see Figure A-3). If they don't show up, select the Windows menu option, then Customize Perspective, and then the Command tab. Ensure that the Tomcat check box in the Available Command Groups is selected.

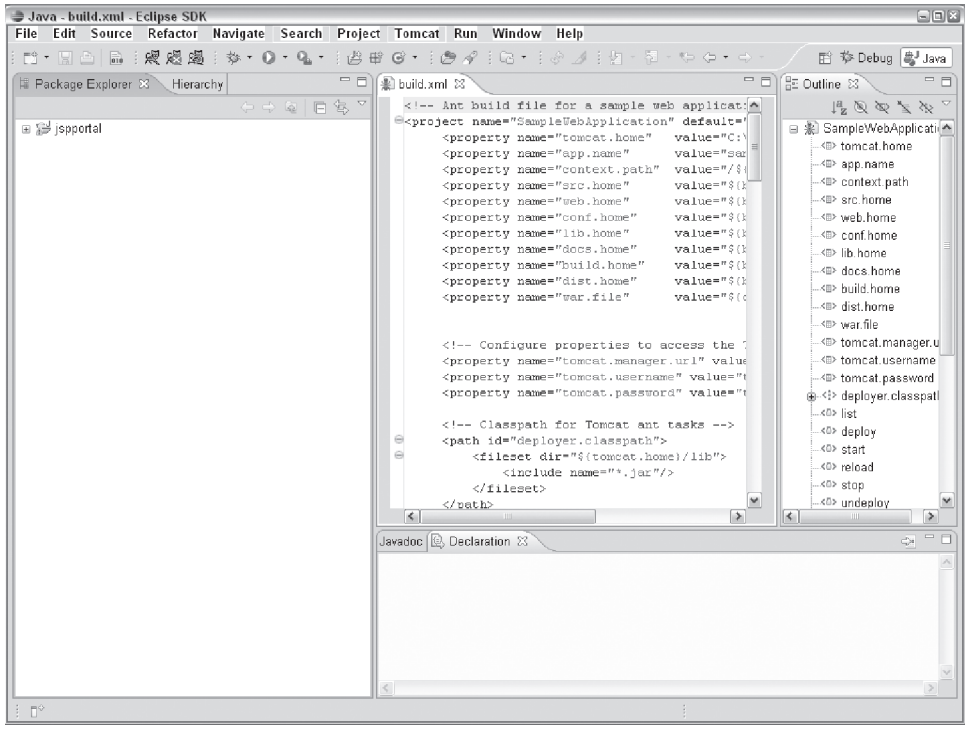

Figure A-3: Tomcat menu options added to Eclipse

4. Now you are ready to configure Tomcat: Select the Windows drop-down menu, and then Preferences, and click on the Tomcat menu option. Configure the version number of Tomcat (Tomcat 6) and the Tomcat home directory (see Figure A-4). Other options allow you to set the JVM and its parameters—remember to select at least Java SE 5 or later—and the Tomcat manager username and password.

5. Click OK.

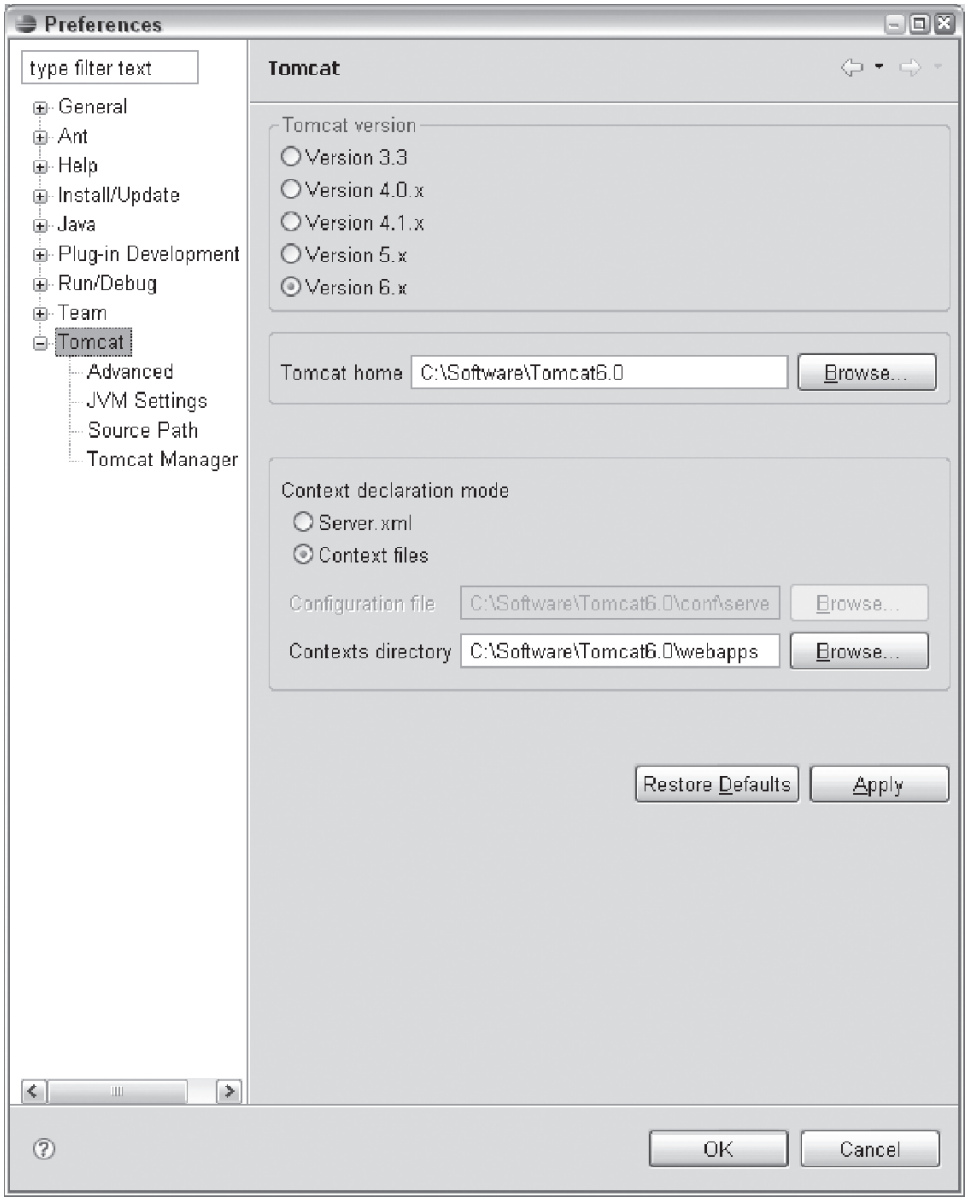

Figure A-4: Configuring Tomcat server properties

Upon completion of these steps, when you start Tomcat, and then debug (or just run) your Web application, the Web application will automatically be deployed to the configured Tomcat instance, and you will be able to step through the execution.

Sysdeo also allows a new type of project to be created — i.e., a Tomcat Project. This wizard eases the Web application development task by creating directory structures appropriate for a Web application, and also adds servlet and JSP jar files to the `CLASSPATH`. Use the following steps to create a Tomcat project:

1. Select the File pull-down menu, followed by Project.
2. Expand the `Java` node and select the Tomcat Project option.
3. Give the project a name and select Next.
4. Click Finish.

Deploying and Debugging Web Applications Using the Web Tools Platform

The Eclipse Web Tools Platform (WTP) provides wizards and applications to help ease the task of development of not just a Web application, but a Web services and Java EE application. WTP can be downloaded from `eclipse.org/webtools/main.php`. The version of WTP used in the section is 2.0M5. We recommend downloading the Web Tools completely, which downloads the WTP and all of its required components.

Installing WTP is straightforward:

1. Download this Zip file, and extract at the same level as the eclipse install directory. This contains both the plugin and features directories, and the files and directories under that should get copied under the Eclipse plugin and features directories, respectively.
2. Launch Eclipse using the command line option `-clean`. This option enables Eclipse to detect a new plugin, and is required only once.

Another way to install WTP is to use the Eclipse update manager to install from the `http://download.eclipse.org/webtools/updates` update site; however, that does not allow for installation of all versions of WTP.

Once WTP is installed, configure the Tomcat server to be used (Window menu option, then Show View, then Other, and finally select the Server category). This brings up the dialog box shown in Figure A-5, which enables you to set the Tomcat server properties.

Now set the server in your project properties to this configured server. Once this is done, you can use the Server tab in your project (see Figure A-6) to start and stop Tomcat, and to debug your Web applications by right-clicking in the Server pane and selecting the appropriate menu item.

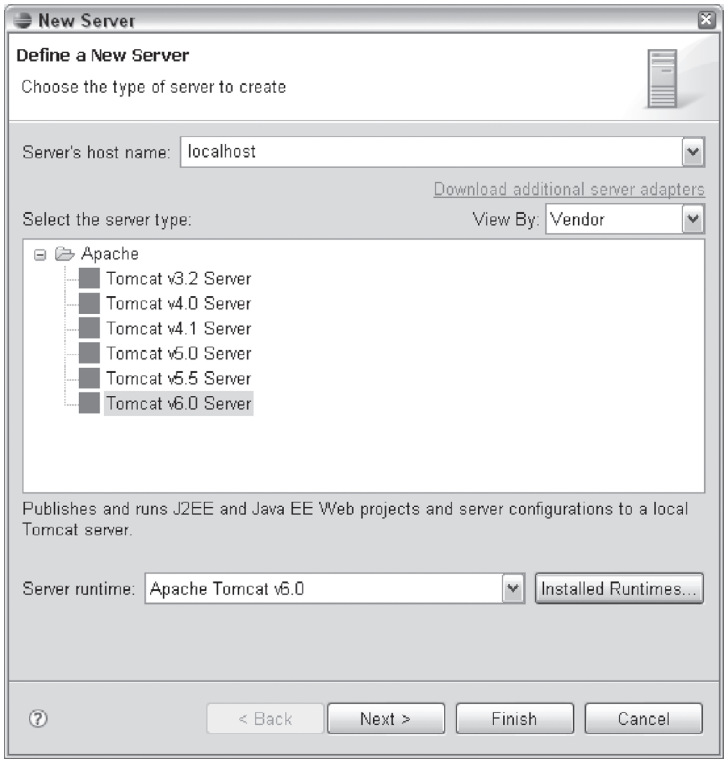

Figure A-5: Configuring the Tomcat server

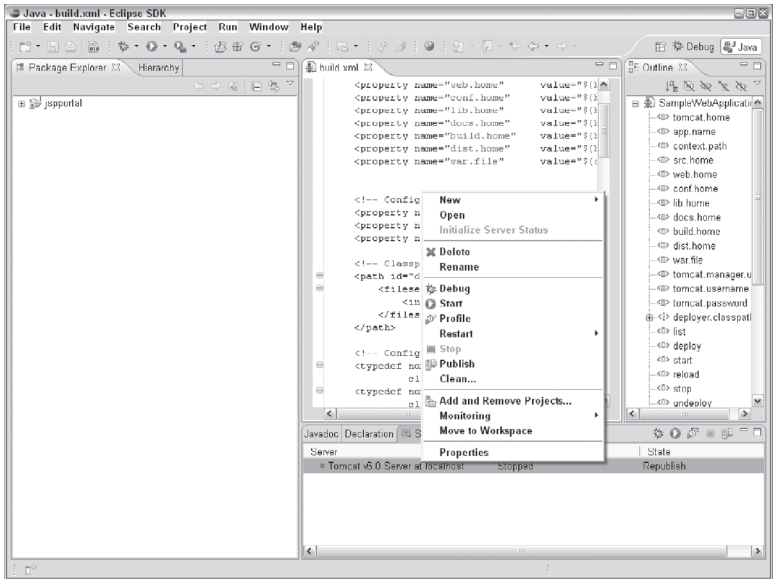

Figure A-6: Managing the Tomcat server from Eclipse

Managing Web Application Deployment Using Apache Ant and Eclipse

Apache Ant is discussed in greater detail in Appendix B, which includes information about how to write Ant tasks for deploying and undeploying Web applications, as well as for starting and stopping Tomcat itself.

Eclipse comes bundled with an Ant plugin, which enables developers to run Ant tasks directly from within the IDE, as shown in Figure A-7.

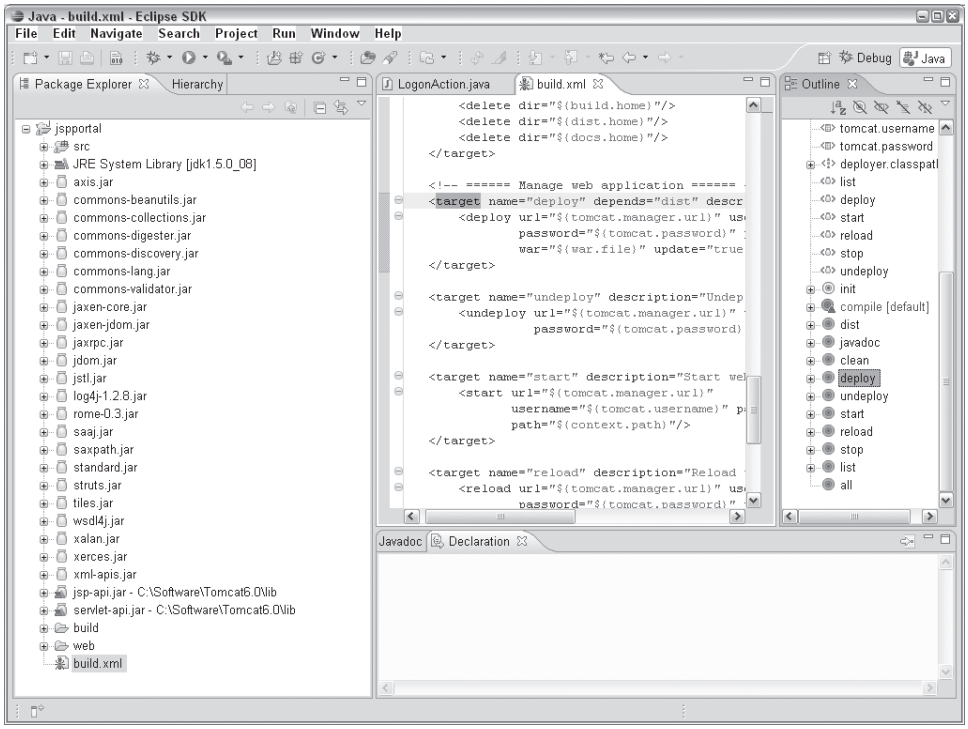

Figure A-7: Running Ant tasks from within Eclipse

As mentioned earlier, this approach allows you to use a common build and deploy script across all deployment environments (development, QA, production).

NetBeans

NetBeans was one of the first Java IDEs, and was developed by Sun Microsystems. It has been available as open source since 2000, and can be downloaded from the `netbeans.org` Web site. The version of NetBeans used in this appendix is 5.5.

Debugging a Remote Web Application in NetBeans

As mentioned earlier, the Java Virtual Machine (JVM) allows for remote debugging. NetBeans, unlike Eclipse, supports both socket and shared memory transports for connecting to the Java process. Naturally, if the process is running on another machine, you can't use the shared memory transport. The network socket transport can be used for both local as well as remote machines.

To enable remote debugging using the network socket transport, you need to do the same two steps: First specify the `JPDA_TRANSPORT` and `JPDA_ADDRESS` to the Tomcat JVM, and second, configure remote debugging parameters in NetBeans.

To specify the JPDA parameters, modify the `catalina.bat` or `catalina.sh` script under `<TOMCAT_HOME>/bin`, and add the following environment variables to the start of the script:

```
JPDA_TRANSPORT=dt_socket
JPDA_ADDRESS=9000
```

Then start Tomcat using the following command:

```
$CATALINA_HOME/bin/catalina jpda start
```

Alternatively, you can set the `JAVA_OPTS` in the `catalina.bat/catalina.sh` script as shown:

```
JAVA_OPTS="-Xdebug -Xrunjdwp:transport=dt_socket,address=9000,server=y,suspend=n";
```

Restart Tomcat after making these changes.

Again, look at Tomcat startup log messages, and find a log message such as `Listening for transport dt_socket at address: 9000`. This indicates that the changes you made have been picked up by Tomcat.

Next, to configure NetBeans, perform the following steps:

1. You should have the Web application's source code configured as an Eclipse project. Set a break point at a location in the Web application code that you wish to debug.
2. Click the Run pull-down menu, and then select the Attach Debugger option. You should now see a dialog box, as shown in Figure A-8.
3. Select the SocketAttach Connector (i.e., `dt_socket Transport`).
4. Specify the IP address or host name of the machine in the `Host` field. This can be `localhost` if Tomcat is running on the same machine as Eclipse.
5. Change the Port field to that specified in the `JPDA_ADDRESS` variable (here `9000`).
6. Click OK.

NetBeans should indicate that it is connecting to the remote Java process. You might sometimes get an error message, such as `Connection refused`. Typical causes of this error include network permissions in connecting to the specified host/port or multiple debugging instances trying to connect to the same Java process.

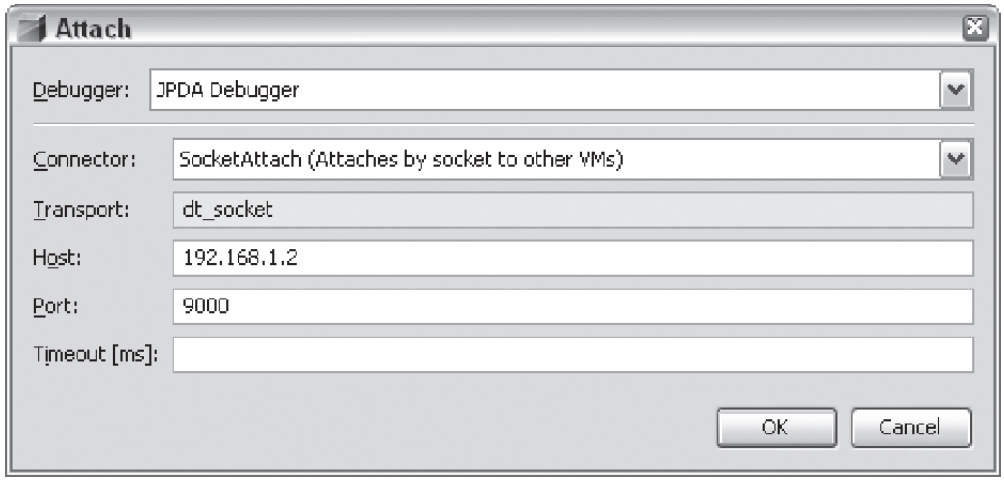

Figure A-8: Configuring the host and port in NetBeans for remote debugging

If everything went well, you can now run the Web application in a browser. Once the code in the Web application reaches the location where you have your break point set, the NetBeans debugger kicks in, and you can debug the remote application (see Figure A-9).

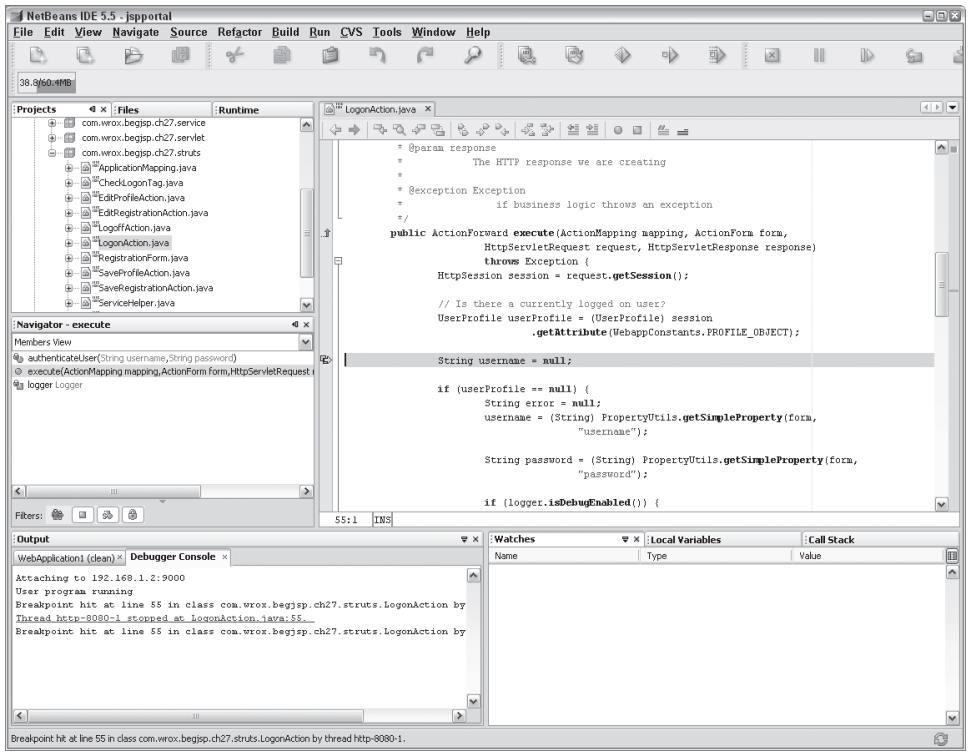

Figure A-9: Debugging a Web application in a remote Tomcat instance

Debugging a Web Application Inside NetBeans

NetBeans includes an embedded version of Tomcat as part of its standard installation. This enables you to execute and debug your Web applications inside the IDE, before deploying to an external Tomcat server.

However, the current version of NetBeans (5.5 at the time of this writing) supports only versions up to Tomcat 5.5, as can be seen in the setup dialog box for a NetBeans Web application project (see Figure A-10).

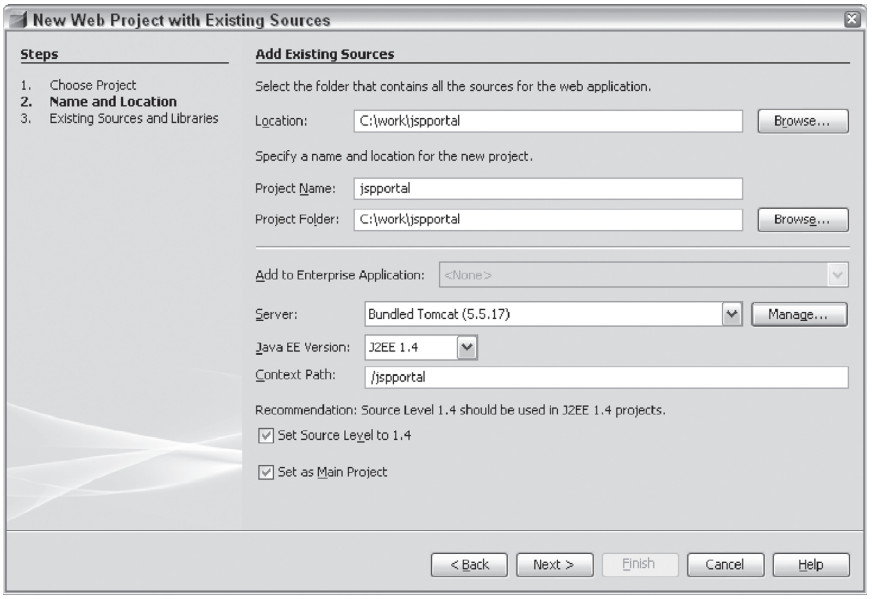

Figure A-10: Debugging a Web application in a remote Tomcat instance

Summary

This appendix does not attempt to compare the features of NetBeans and Eclipse, or that of any other IDE. Most IDEs have a similar set of core features, and sometimes developers have a strong preference for one over the others.

The appendix covered the following:

- Debugging Web applications in the Eclipse and the NetBeans IDE.
- Debugging remote Web applications.
- Managing the Tomcat server (start, stop, restart) from within the IDE itself.
- Using Apache Ant support in both IDEs, which enables you to run Ant tasks from within the IDE. This approach is recommended because it promotes the use of common build and deploy scripts across build environments, thus minimizing potential errors.

These features, while very useful in a development or test environment, should never be used against a production deployment.

Appendix B provides a tutorial introduction to Apache Ant.

Apache Ant

Ant is the de facto standard for creating cross-platform build scripts for Java applications. An important feature that led to its popularity is the capability it offers developers to extend it via custom tasks. In Chapter 8, you learned how Tomcat's custom Ant tasks can be used to deploy and undeploy Web applications without the need to restart Tomcat. This means that not only can you build your application using an Ant build script, but you can also go a step further by installing, removing, reloading, or monitoring the Web application from the same build script. You can do a lot more with Ant, as this appendix illustrates.

This appendix provides a comprehensive overview of Ant features and capabilities. It covers the following:

- A short, tutorial-style introduction to Ant.
- Solutions ("recipes") for common tasks that system administrators need to do while developing build and deploy systems. Some of these tasks include:
  - A basic build script for a Web application
  - Making this build script reusable for different environments using property files and command-line parameters
  - Sending notifications of successful (or failed) builds to developers via e-mail
  - Ant and source code control systems
  - Further references for setting up a continuous integration environment
- A reference for common Ant tasks.

Installing Ant

Ant started off as a subproject under the Jakarta project. Since then, its increasing popularity earned it a promotion — it's now a top-level project under Apache, and can be downloaded from `http://ant.apache.org`. This appendix uses the latest stable release available (currently Ant 1.7.0).

Installing Ant is simple:

1. Download Ant from `www.apache.org` and unzip it in a directory of choice. For the remainder of the appendix, `$ANT_HOME` (`%ANT_HOME%` for Windows) is used as the environment variable that points to the installation directory of Ant.
2. Add `$ANT_HOME/bin` to your system `PATH`.

On Unix/Linux:

```
$ ANT_HOME=/path/to/ant
$ PATH=$ANT_HOME:$PATH; export PATH
```

On Windows, use the Control Panel (System➪Advanced➪Environment Variables) to add `%ANT_HOME%\bin` to your PATH variable.

Ant is often used to perform additional tasks, other than building Java code. Typically, this requires that you copy JAR files for these custom tasks into `$ANT_HOME/lib`.

For example, when Ant is to be used to manage Web applications, you need to copy the `catalina-ant.jar` file that contains Tomcat's Ant tasks from the `$CATALINA_HOME/lib` directory to `$ANT_HOME/lib`. The `catalina-ant.jar` jar file was located in the `$CATALINA_HOME/server/lib` directory in all Tomcat versions prior to Tomcat 6.

Introduction to Ant

As a system administrator, you have likely been exposed to a lot of build tools (`make`, `jam`, and so on), so why is another build tool required?

Ant is built around the following central ideas:

- Implement the tool using Java, and use XML to store the build information. This results in a platform-independent build tool.
- Enable extensibility of the tool. Developers can extend Ant by writing Java classes, and thus develop custom tasks. One example of this is Tomcat's management tasks, mentioned earlier. Another example of this kind of integration is the capability to run JUnit test cases from Ant build scripts, using the optional `<junit>` task.

The first thing most people miss when moving from `make` to Ant is the expressiveness of `make`. Make-like tools are based on the underlying shell, and while that is a very powerful tool and leads to compact build scripts, such scripts are nonportable. However, if you absolutely need to execute a shell command, Ant does offer a way out. Ant's `<exec>` task allows for this at the cost of the portability of build scripts.

As mentioned earlier, Ant uses an XML file to store build information. This file, called `build.xml` by default, contains the list of tasks to be performed. The general structure of an Ant build file is as shown in Figure B-1.

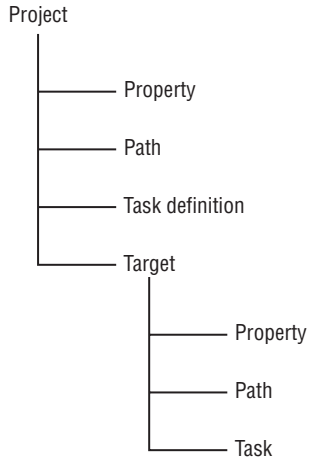

Figure B-1: General structure of an Ant script

A project consists of a number of property settings, targets, paths, and task definitions.

Properties at the project level are name-value pairs that are available throughout the project and to each target.

A `target` consists of a series of tasks. A target can define its own set of properties, which override the global project properties. A target can depend on other targets, which means that all targets that it depends upon will execute first, before running the tasks associated with it. Ant comes with several built-in tasks that can be called. Some of the built-in tasks include creating directories, copying files, compiling Java source files, and so on.

You can also define path elements at both the project level and target level. A path is used to include or exclude certain files and directories. For example, you can construct a path element to contain the directories and/or JAR files that constitute the classpath.

Let's take a look at a simple Ant file (`FirstBuild.xml`). This build file creates a directory and then copies a file to that directory.

First, the `<project>` element must be specified:

```
<!--
  A simple build script that creates a directory (dir.name), and copies
  a file (file.name) to it.
-->
<project name="MyFirstAntProject" basedir="." default="copyfile">
```

The `name` attribute in the `<project>` element is set to the name of the project (`MyFirstAntProject` in this case). The `basedir` attribute indicates the root directory, which will be used as a reference for all the tasks present in this project. The `default` attribute indicates the target that will be executed by default if no attributes are specified while running Ant. Here, it is set to the `copyfile` target, which is covered later in the section. Note also the comments: as in all XML (and HTML) documents, anything between the `<!-- and -->` delimiters is taken as a comment.

Next, the properties for the project are defined:

```
<property name="dir.name" value="${basedir}/mydir"/>
    <property name="file.name" value="file1.txt"/>
```

Here, two global properties are defined: `dir.name` and `file.name`. The `dir.name` property specifies the name of the directory to be created, and `file.name` is the file to be copied. The properties are optional: The names of the directory and file to be created can be specified directly in the target itself. However, as you will see later, defining properties allows for reusable scripts as these can be overridden through property files, or from the command line. Also, if these values are used in multiple places, it makes for easier, and less error prone, modifications.

Because this is a very simple Ant script, it has no path and task definitions.

Finally, the targets to be executed are specified. In this project, these include creating a directory and copying the file into the newly created directory:

```
    <target name="makedirectory" description="Create directory mydir">
          <mkdir dir="${dir.name}"/>
    </target>
    <target name="copyfile" depends="makedirectory" description="Copy files">
          <copy file="${file.name}" todir="${dir.name}"/>
    </target>
    <target name="clean" description="Clean up task">
          <delete dir="${dir.name}"/>
    </target
</project>
```

Note the `depends` attribute for the `copyfile` target. It indicates that `copyfile` is dependent on `makedirectory`. Therefore, even if you specify the `copyfile` target alone, Ant will make sure that all the dependencies are run first: i.e., `makedirectory` will be executed first, and then `copyfile`.

The target `makediretory` creates the directory. Note how the directory name is referenced via the `${dir.name}` property. The built-in tasks `<mkdir>` and `<copy>` are used to perform the functions of making a directory and copying the file. The syntax of the Ant command is as follows:

```
ant -buildfile <filename> <target-name>
```

If the `buildfile` option is not used, Ant will look for a file named `build.xml` in the directory from which the Ant command was issued.

```
$ ant
Buildfile: build.xml does not exist!
Build failed
```

In this case, the build script is not named `build.xml` — hence the previous error.

The following example shows the Ant command being run with the `FirstBuild.xml` build file:

```
$ ant -buildfile FirstBuild.xml
Buildfile: FirstBuild.xml
makedirectory:
    [mkdir] Created dir: /home/tomcat/AppendixB/mydir
copyfile:
     [copy] Copying 1 file to /home/tomcat/AppendixB/mydir
BUILD SUCCESSFUL
Total time: 1 second
```

If the target name is not specified, Ant will look for the default target to execute as specified by the `default` attribute of the root `<project>` element.

As mentioned earlier, the values of the properties (`dir.name` and `file.name`) can be overridden from the command line. This is illustrated here:

```
$ ant -buildfile FirstBuild.xml -Ddir.name=yourdir
Buildfile: FirstBuild.xml
makedirectory:
    [mkdir] Created dir: /home/tomcat/AppendixB/yourdir
copyfile:
     [copy] Copying 1 file to /home/tomcat/AppendixB/yourdir
BUILD SUCCESSFUL
Total time: 1 second
```

By convention, you should define a target called `clean` that does any cleanup required for the project. In this case, the `clean` target just undoes the create directory operation by deleting the directory.

```
$ ant -buildfile FirstBuild.xml clean
Buildfile: FirstBuild.xml
clean:
   [delete] Deleting directory /home/tomcat/AppendixB/mydir
BUILD SUCCESSFUL
Total time: 0 seconds
```

More Command-Line Options

Finally, here are a few of the Ant command-line options that can be very useful:

- `-projecthelp`
- `-verbose`
- `-debug`
- `-logfile`
- `-keep-going`

To see what targets are available in an Ant script, use the -projecthelp option, as shown:

```
$ ant -f FirstBuild.xml  -projecthelp
Buildfile: FirstBuild.xml
Main targets:
 clean          Clean up task
 copyfile       Copy files
 makedirectory  Create directory mydir
Default target: copyfile
```

The -debug option shows a debug output of the build, such as what the CLASSPATH and other environment variable settings are, and can be quite useful for debugging issues with build scripts. Another option useful for understanding what a build script does is the -verbose option that, as the name suggests, shows a verbose output of the Ant script execution.

The -logfile option specifies a log file where the output of the Ant script execution is logged.

The -keep-going option tells Ant to keep executing even if an error occurs while running the script. The default behavior of Ant is to abort on any error.

Finally, the -help option prints out all the command-line options supported by Ant.

Ant Recipes

Now that you are familiar with Ant concepts, it is time to move on to more real-world tasks. Often as a system administrator, you are asked to design, build, or modify build scripts for multiple environments. Toward that end, this section presents recipes for some common tasks that need to be performed. Some of these tasks are:

- A basic build script for a Web application.
- Making this build script reusable for different environments using property files and command-line parameters. The multiple environments may be either different operating systems (Windows, Linux/Unix) or different deployment environments (development, QA, staging, production).
- Recording a log of the build script execution.
- Sending notifications of successful (or failed) builds to developers via e-mail.
- Integrating Ant with source code control systems: Ant scripts often need to access source code stored in version control systems such as CVS or ClearCase.
- Setting up an automated system for builds and deployment.
- Further references for setting up a continuous integration environment. Agile development techniques place a lot of emphasis on such features, and while this is a huge topic, some preliminary techniques are covered here.

Building Web Applications with Ant

This section demonstrates how to build a sample Web application with Ant. The steps include compiling the files and creating the appropriate directory structure for the WAR file to get the application ready for deployment.

A sample development-time directory structure for a Web application project may look like the directory structure shown in Figure B-2.

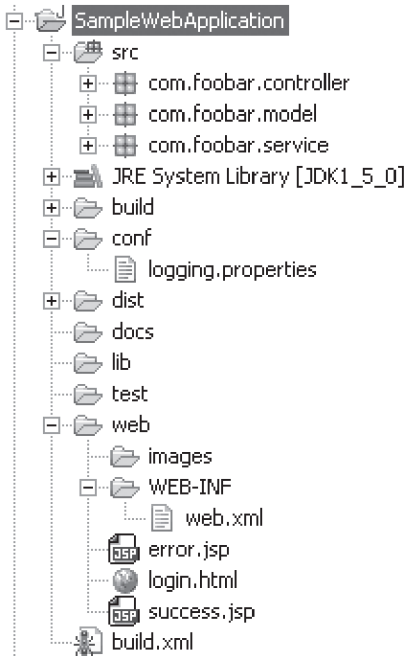

Figure B-2: Sample source code organization for a Web application project

This directory structure consists of the following:

- The main build file (`build.xml`).
- A directory (`src`) containing all the Java source files of the Web application (for example, all the Servlet classes and Java beans).
- A directory (`web`) containing the HTML and JSP files. It also contains any other resource directories — for example, images. Finally, it contains the `WEB-INF` directory with the deployment descriptor (`web.xml`). Chapter 7 covers the deployment descriptor for the Web application in detail. The `WEB-INF` directory also has a `lib` directory that contains any third-party JAR files.
- A directory (`build`) in which the compiled Java classes would be built.
- The deployable WAR file is generated in the `dist` directory.
- The Javadocs for the project are generated in the `doc` directory. The `build`, `dist`, and `doc` directories are created by the build script.

The names of the directories (`build`, `dist`, `src`, and so on) are merely development conventions, and not anything required by Ant.

The `build.xml` build file is shown here (the default target is the `compile` target):

```
<!-- Ant build file for a sample web application -->
<project name="SampleWebApplication" default="compile" basedir=".">
```

The following section initializes the global properties that are used throughout the build file. Note that you might have to change some of these properties to suit your environment. For example, the `tomcat.home` property should point to the root directory of your Tomcat installation:

```
<property name="tomcat.home"        value="/usr/tomcat/apache-tomcat-6 "/>
<property name="app.name"           value="sampleWebapp"/>
<property name="context.path"       value="/${app.name}"/>
<property name="src.home"           value="${basedir}/src"/>
<property name="web.home"           value="${basedir}/web"/>
<property name="conf.home"          value="${basedir}/conf"/>
<property name="lib.home"           value="${basedir}/lib"/>
<property name="docs.home"          value="${basedir}/docs"/>
<property name="build.home"         value="${basedir}/build"/>
<property name="dist.home"          value="${basedir}/dist"/>
<property name="war.file"           value="${dist.home}/${app.name}.war"/>

<!-- Configure properties to access the Tomcat Manager application -->
<property name="tomcat.manager.url" value="http://localhost:8080/manager"/>
<property name="tomcat.username"    value="tomcat"/>
<property name="tomcat.password"    value="tomcat"/>
```

In this build file, the properties are included in the file itself for the sake of simplicity. However, a good development practice is to move them to a separate properties file, as this allows for reusable build scripts. This practice is covered later in the chapter in the section "Reusable Ant Scripts Using Property Files and Command-Line Parameters."

The `clean` target deletes the `build` and `dist` directories and all subdirectories within them. This target is useful if you want to remove all the files generated by a build:

```
<!-- ====== Clean Target ====== -->
<target name="clean"
         description="Cleanup- deletes everything generated by the ant script">
     <delete dir="${build.home}"/>
     <delete dir="${dist.home}"/>
     <delete dir="${docs.home}"/>
</target>
```

The `init` target does all the initialization required, such as creating the directory structure, and initializing the Java `CLASSPATH`. The `CLASSPATH` includes all libraries required for the Web application itself, as well as Tomcat JAR files required for compiling the servlet code. As shown in the listing that follows, it includes the `servlet-api.jar` file that is required for compiling servlets. This JAR file is located under `$CATALINA_HOME/lib`. In Tomcat versions prior to Tomcat 6, this JAR file was in `$CATALINA_HOME/common/lib`.

```
<!-- ====== All initializations: Classpath, directory structure ====== -->
<target name="init">
```

```
    <mkdir  dir="${build.home}"/>
    <mkdir  dir="${docs.home}"/>
    <mkdir dir="${dist.home}"/>
    <!-- Classpath for compiling web application, generating javadocs -->
    <path id="classpath">
       <fileset dir="${lib.home}">
           <include name="*.jar"/>
       </fileset>
       <fileset dir="${tomcat.home}/lib">
           <include name="servlet-api.jar"/>
       </fileset>
    </path>
    <property name="classpath" refid="classpath"/>
</target>
```

The `compile` target compiles all the Java source files present in the `src` directory. The `CLASSPATH` settings done earlier in the `init` target are specified here using the `classpath` attribute. The class files are output in the directory pointed to by the `destdir` attribute (set to the `build.home` directory):

```
<!-- ====== Compilation  ====== -->
<target name="compile" depends="init">
    <echo message="Classpath set to ${classpath}"/>
    <javac srcdir="${src.home}"
           destdir="${build.home}"
           debug="true"
           classpath="${classpath}"
           deprecation="true">
    </javac>
    <!-- Copy all property files -->
    <copy todir="${build.home}">
        <fileset dir="${conf.home}"/>
    </copy>
</target>
```

The `dist` target creates a WAR file that can be deployed in the Tomcat container:

```
<!-- ====== Create a distributable WAR file ====== -->
<target name="dist" depends="compile"
        description="Creates the deployable WAR file">
    <war destfile="${war.file}"
        webxml="${web.home}/WEB-INF/web.xml">
        <fileset dir="${web.home}" excludes="**/web.xml" />
        <lib dir="${lib.home}"/>
        <classes dir="${build.home}"/>
    </war>
</target>
```

The `javadoc` target creates the documentation for the source code. Note the use of the `CLASSPATH` through the `classpath` attribute, which was initialized earlier in the `init` target:

```
<!-- ====== Create documentation (javadocs) ====== -->
<target name="javadoc" depends="init" description="Creates the Javadocs for the
project">
```

(continued)

```
    <javadoc sourcepath="${src.home}"
             packagenames="com.foobar.*"
             classpath="${classpath}"
             destdir="${docs.home}"
             windowtitle="Javadoc for the Sample Web Application (TM)">
    </javadoc>
</target>
```

The `all` target runs all the targets in the order specified in the `depends` attribute for this target:

```
<!-- ====== All Target ====== -->
<target name="all"
        depends="clean, prepare, compile, dist"
        description="Builds the web application and war file"/>
```

Let's run the different targets now to ensure that your environment is set up to run Ant correctly.

- `clean`: Open the console window and go to the `/home/tomcat/AppendixB` directory and run the `clean` target as shown here. Note that if you run the `clean` target after running the `compile` or `dist` target, the `build` and `dist` directories will be cleared in the `clean` target:

```
$ant clean
Buildfile: build.xml
clean:
   [delete] Deleting directory /home/tomcat/AppendixB/build
   [delete] Deleting directory /home/tomcat/AppendixB/dist
BUILD SUCCESSFUL
Total time: 2 seconds
```

- `dist`: The `dist` target is responsible for generating the WAR file. Because this target depends on the `compile` target, by running it you ensure that the Java file is also compiled.

```
$ ant dist
Buildfile: build.xml
init:
    [mkdir] Created dir: /home/tomcat/AppendixB/build
    [mkdir] Created dir: /home/tomcat/AppendixB/docs
    [mkdir] Created dir: /home/tomcat/AppendixB/dist
compile:
    [javac] Compiling 4 source files to /home/tomcat/AppendixB/build
     [copy] Copying 3 files to /home/tomcat/AppendixB/build
dist:
      [war] Building war: /home/tomcat/AppendixB/dist/sampleWebapp.war
BUILD SUCCESSFUL
Total time: 5 seconds
```

Now that you have the expanded WAR directory structure for the Web application as well as the `.war` file, you are ready to deploy it. This Web application can be deployed in a number of ways:

- Copy the WAR file to the `$CATALINA_HOME/webapps` directory.
- Create a context for the Web application by making a directory within `$CATALINA_HOME/webapps` — for example, `$CATALINA_HOME/webapps/sampleWebapp` — and copy the expanded WAR directory structure in the `build` directory to `$CATALINA_HOME/webapps/sampleWebapp`.

- Use the `manager` Web application GUI to deploy the application.
- Use the Ant interface to the `manager` application, as shown earlier in Chapter 8.

The Ant targets for managing the Web application and the properties that need to be configured to use them are listed here:

```
<!-- Configure properties to access the Tomcat Manager application -->
<property name="tomcat.manager.url" value="http://localhost:8080/manager"/>
<property name="tomcat.username" value="tomcat"/>
<property name="tomcat.password" value="tomcat"/>
<!-- Classpath for Tomcat ant tasks -->
<path id="deployer.classpath">
        <fileset dir="${tomcat.home}/lib">
            <include name="*.jar"/>
        </fileset>
</path
...
<!-- ====== Manage web application ====== -->
<target name="deploy" depends="dist" description="Deploy web application">
        <deploy url="${tomcat.manager.url}" username="${tomcat.username}"
                password="${tomcat.password}" path="${context.path}"
                war="${war.file}" update="true" />
</target>
<target name="undeploy" description="Undeploy web application">
        <undeploy url="${tomcat.manager.url}" username="${tomcat.username}"
                  password="${tomcat.password}" path="${context.path}"/>
</target>
<target name="start" description="Start web application">
        <start url="${tomcat.manager.url}" username="${tomcat.username}"
               password="${tomcat.password}" path="${context.path}"/>
</target>
<target name="reload" description="Reload web application">
        <reload url="${tomcat.manager.url}" username="${tomcat.username}"
               password="${tomcat.password}" path="${context.path}"/>
</target>
<target name="stop" description="Stop web application">
        <stop url="${tomcat.manager.url}" username="${tomcat.username}"
              password="${tomcat.password}" path="${context.path}"/>
</target>
<target name="list" description="List all web applications on server">
        <list url="${tomcat.manager.url}" username="${tomcat.username}"
              password="${tomcat.password}"/>
</target>
```

Once the Web application is built, it can be deployed using the command:

```
$ ant deploy
```

The other targets (`undeploy`, `start`, `stop`, and `reload`), as the names suggest, allow for undeploying, starting, stopping, and reloading the Web application.

Compiling JSPs

In development environments JSPs are usually not compiled while deploying the Web application. This allows a very rapid development cycle, as the JSP pages can be modified and their effect seen immediately by reloading the Web page in the browser. Behind the scenes, Tomcat does the work of detecting if a JSP has changed and if so, generating Java code for it (a servlet), compiling and then loading the class file.

While this is great for development, making Tomcat monitor the JSPs for modification has a performance impact. A common practice in production environments is to precompile JSPs and have Tomcat configured to not check for modified JSP files.

Precompiling JPSs requires two steps in your build script: First convert the JSP to Java code using the `jspc` task, and then have the `javac` task compile the generated servlets. These steps are shown in the highlighted code. Note the use of Tomcat's `jspc` task (`org.apache.jasper.JspC`) in the `taskdef`. Ant comes with a `jspc` task too, but there's a bug in the way it works with Tomcat.

```
...
<taskdef name="jasper" classname= "org.apache.jasper.JspC">
    <classpath>
        <fileset dir="${tomcat.home}/lib">
            <include name="jasper.jar"/>
            <include name="jasper-el.jar"/>
            <include name="servlet-api.jar"/>
            <include name="jsp-api.jar"/>
            <include name="el-api.jar"/>
        </fileset>
        <fileset dir="${tomcat.home}/bin">
            <include name="tomcat-juli.jar"/>
        </fileset>
    </classpath>
</taskdef>
<!-- ====== All initializations: Classpath, directory structure ====== -->
<target name="init">
    <mkdir  dir="${build.home}"/>
    <mkdir  dir="${docs.home}"/>
    <mkdir dir="${dist.home}"/>
    <!-- Classpath for compiling web application, generating javadocs -->
    <path id="classpath">
        <fileset dir="${lib.home}">
            <include name="*.jar"/>
        </fileset>
        <fileset dir="${tomcat.home}/lib">
            <include name="jasper.jar"/>
            <include name="jasper-el.jar"/>
            <include name="servlet-api.jar"/>
            <include name="jsp-api.jar"/>
            <include name="el-api.jar"/>
        </fileset>
    </path>
    <property name="classpath" refid="classpath"/>
</target>
```

```
<!-- ====== Compilation  ====== -->
<target name="compile" depends="init">
    <jasper uriRoot="${web.home}"
            outputdir="${src.home}">
    </jasper>
    <echo message="Classpath set to ${classpath}"/>
    <javac srcdir="${src.home}"
           destdir="${build.home}"
           debug="true"
           classpath="${classpath}"
           deprecation="true">
    </javac>
    <!-- Copy all property files -->
    <copy todir="${build.home}">
        <fileset dir="${conf.home}"/>
    </copy>
</target>
```

Reusable Ant Scripts Using Property Files and Command-Line Parameters

As mentioned earlier, a good development practice is to move all configurable properties from the build script to a separate properties file. This property file would contain all details that vary from project to project, such as the name of the Web application, `CLASSPATH` setting, Tomcat manager URL, and so on.

This enables you to use the same build script for different environments. The property file can then be specified via a command-line option:

```
ant -buildfile <filename> <target-name> -propertyfile
<propertyfilename>
```

In the subsequent sections, you will see some tips on how this parameterization of the Ant script can be done. In addition to being a good development practice, this also helps minimize errors that having a different build script for each environment can introduce.

Reusable Scripts for Different Operating Systems (Linux/Unix, Windows)

In spite of the cross-platform nature of Java and Web applications, there are sometimes differences when building and deploying on different operating systems. For example, the location of the log file directory, temporary directory, and so on would be different for Windows and Linux.

Ant has an internal property called `os.name` that evaluates to the name of the operating system, such as "Linux" or "Windows XP," depending on where the Ant script is run. This can be used to conditionally include different property files based on the platform the build script is run on. The following example shows this, and the Ant script includes either `Linux.properties` or `Windows XP.properties`, depending on whether it is run on Linux or Windows XP.

```
<property file="${os.name}.properties"/>
```

Following is another example of this, where there are multiple versions of the `logging.properties` files (the property file used by the Java Logging API) for each of the operating system platforms the script is run on:

```
<!-- Copy OS specific log property file -->
<echo message="OS is '${os.name}'"/>
<copy tofile="${build.home}/logging.properties"
      file="${conf.home}/logging.properties-${os.name}"/>
```

Test the name returned by the `os.name` *property using* `<echo>` *before deciding on the file name for your property files.*

Reusable Scripts for Different Environments: Development, QA, Staging, and Production Systems

Most enterprise-level projects have multiple build environments: development environments used by software developers, QA environments used for testing, and staging environments where the tested application is "staged" before being deployed to production. The kind of things that change between these deployments are database property files (production, staging, QA, and development often have different databases that are isolated from each other), deployment server URLs, and so on.

A technique similar to that used for different operating system environments can be used here, too, by defining a custom Ant property. In the following example, property files are defined for each development environment (`dev.properties`, `qa.properties`, `staging.properties`, and `production.properties`). One of these files is then included based on the value of the `environment` property.

```
<!-- Check if the "environment" property is set -->
<fail
    message="Environment not set: must be one of dev/qa/production/staging">
    <condition>
        <not>
            <isset property="environment"/>
        </not>
    </condition>
</fail>
<!-- Include the property file for the appropriate environment -->
<echo message="Building for ${environment} environment."/>
<property file="${environment}.properties"/>
```

The `environment` property is passed to the build script via the `-D` command-line parameter, as shown:

```
ant -buildfile <filename> <target-name> -Denvironment=qa
```

Build Logs

The output of the build script goes to the console. Often an administrator has to run a lot of builds in a day, and needs a way to keep a log of the build or at least critical portions of the build script execution. Ant's `record` task is a neat way to do this. It allows you to record the results of a build, and turn the recording on and off in portions of the build script that are of interest. The following changes to the build script demonstrate the use of the `record` task. As you can see from the example, the log from the build script execution is appended to the `build.log` file.

```
<!-- ====== Compilation  ====== -->
<target name="compile" depends="init">
    <record name="build.log" action="start"
            loglevel="verbose" append="yes"/>
    <echo message="Classpath set to ${classpath}"/>
    <javac srcdir="${src.home}"
           destdir="${build.home}"
           debug="true"
           classpath="${classpath}"
           deprecation="true">
    </javac>
    <!-- Copy all property files -->
    <copy todir="${build.home}">
        <fileset dir="${conf.home}"/>
    </copy>
         <record name="build.log" action="stop"/>
</target>
<!-- ====== Create a distributable WAR file ====== -->
<target name="dist" depends="compile"
        description="Creates the deployable WAR file">
    <record name="build.log" action="start"
            loglevel="verbose" append="yes"/>
    <war destfile="${war.file}"
         webxml="${web.home}/WEB-INF/web.xml">
        <fileset dir="${web.home}" excludes="**/web.xml" />
        <lib dir="${lib.home}"/>
        <classes dir="${build.home}"/>
    </war>
    <record name="build.log" action="stop"/>
</target>
```

Another way to log the output of an Ant build is to specify a log file using the `-logfile` command-line option. This would take everything output by Ant on the command line and write it to the specified log file.

Build Notifications via E-mail

Developers often need to know the status of a build. If the Ant build script is run manually, the results can be e-mailed by the person running the build process. In big projects, or those with distributed development, it is common to have builds fired off automatically. This section describes how e-mail notifications of build status can be generated and automatically e-mailed.

Ant enables you to monitor the status of a build using listeners and loggers. The listeners are components that enable the monitoring of Ant events, such as the start and end of a task, a target, or a build. The loggers extend the functionality of listeners and are responsible for logging information about a build.

A `logger` class can be associated with a build process using Ant's `-logger` command-line option:

```
ant -logger <loggername>
```

Here *<loggername>* is the fully qualified class name of the `logger` class.

Similarly, a `listener` class can be associated with the build process using the `-listener` command-line option:

```
ant -listener <listenername>
```

Ant provides a built-in class called `MailLogger` (`org.apache.tools.ant.listener.MailLogger`) that can be used to e-mail results of the build process. You can associate this `logger` with a build process by using the following command:

```
ant -logger org.apache.tools.ant.listener.MailLogger
```

When the build file finishes executing, the `logger` class sends an e-mail about the build's status. The behavior of the `logger` can be controlled via several properties, as described in the following table.

Property Name	Description
`MailLogger.mailhost`	The outgoing SMTP mail server that is used to send the e-mail. This property is mandatory.
`MailLogger.from`	The e-mail address of the account from which the e-mail is sent. This property is mandatory.
`MailLogger.failure.notify`	This Boolean property indicates whether an e-mail notification must be sent in case the build fails. This property is optional and is enabled by default.
`MailLogger.success.notify`	This Boolean property indicates whether an e-mail notification must be sent if the build succeeds. If you are interested in sending an e-mail message only when there is a failure, you can set this property value to `false`. This property is optional and has a default value of `true`.
`MailLogger.failure.subject`	The subject of the e-mail in case the build fails. This property is optional and its default value is `Build Failure`.
`MailLogger.failure.to`	The e-mail address to which the build results must be sent in case of a failure. You can send the results to multiple e-mail addresses by separating them with commas. This property is needed only if you need to send an e-mail because of a failure.
`MailLogger.success.subject`	The subject of the e-mail if the build succeeds. This property is optional and its default value is `Build Success`.
`MailLogger.success.to`	The e-mail address to which the build results must be sent in case of success. You can send the results to multiple e-mail addresses by separating them with commas. This property is mandatory only if you need to send an e-mail if the build is successful.

Following is a sample properties file (`MailLogger.properties`):

```
MailLogger.mailhost=<your-smtp-servername>
MailLogger.from=<youraccount@someserver.com>
MailLogger.failure.subject=BUILD FAILURE : My Intranet Application
MailLogger.failure.to=<youraccount@someserver.com>
MailLogger.success.subject=BUILD SUCCESSFUL : My Intranet Application
MailLogger.success.to=<youraccount@someserver.com>
```

The property file is specified on the command line via the `-propertyfile` attribute:

```
$ ant dist -logger org.apache.tools.ant.listener.MailLogger -propertyfile
MailLogger.properties
```

In addition to the `MailLogger`, the other loggers and listeners available are as follows:

- **DefaultLogger (`org.apache.tools.ant.DefaultLogger`):** This is the default Ant logger, and prints build-related messages to the console.
- **NoBannerLogger (`org.apache.tools.ant.NoBannerLogger`):** This logger functions similarly to the `DefaultLogger`, only it doesn't output messages for targets that don't perform any action.
- **AnsiColorLogger (`org.apache.tools.ant.listener.AnsiColorLogger`):** This logs the same messages that the `DefaultLogger` does, but in color.
- **Log4jListener (`org.apache.tools.ant.listener.Log4jListener`):** This is a listener that passes events to Log4J.
- **XMLLogger (`org.apache.tools.ant.XmlLogger`):** This logs messages in an XML format to a log file specified by the `-logfile` command-line option.

Developers can also write their own loggers/listeners by implementing the `org.apache.tools.antBuildListener` Java interface.

Ant and Source Control Systems

Source code is often checked into version control systems such as CVS. You could always check out the code, and then run the build script. However, Ant has tasks for interacting with CVS, and so this task, too, could be automated as shown:

```
<!-- ====== Compilation  ====== -->
<target name="compile" depends="init">
     <cvs cvsRoot=":pserver:user1@cvs.foobar.com:/home/cvs"
          package="sampleWebappSrc"
          dest="${src.home}"
     />
     <echo message="Classpath set to ${classpath}"/>
     <javac srcdir="${src.home}"
            destdir="${build.home}"
            debug="true"
            classpath="${classpath}"
            deprecation="true">
     </javac>
     <!-- Copy all property files -->
```

```
        <copy todir="${build.home}">
            <fileset dir="${conf.home}"/>
        </copy>
    </target>
```

Details of Ant tasks for other source control systems such as Perforce, ClearCase, SourceSafe, and others can be found at `http://ant.apache.org/manual/tasksoverview.html#scm`.

Automated Testing

You can add a target to do unit testing as shown in the code that follows. This target ensures that only code that passes all unit tests gets deployed.

The JUnit task is not a part of the Ant core tasks, and therefore you must get the `junit.jar` *file from* `www.junit.org` *and copy it to* `$ANT_HOME/lib` *before running this script.*

If you are running your Ant script from the Eclipse IDE, you already have a JUnit plug-in installed. All you need to do is tell Ant where to find it: Select Window➪Preferences and then Ant➪Runtime. In the `Classpath` *tab, click first on Global Entries and then Add External JARs. Select the* `junit jar` *that is inside your* `<ECLIPSE_HOME>/plugin/org.junit_x.y.z` *directory.*

```
<!-- Ant build file for a sample web application -->
<project name="SampleWebApplication" default="compile" basedir=".">
    ...
    <property name="test.home" value="${basedir}/test"/>
    ...
    <!-- ====== Compilation  ====== -->
    <target name="compile" depends="init">
    ...

    </target>
    <target name="unit-test"
            depends="compile"
            description="Runs all unit test">
        <junit printsummary="yes" haltonfailure="yes">
            <classpath>
                <pathelement location="${build.home}"/>
                <pathelement path="${classpath}"/>
            </classpath>
            <batchtest fork="yes" todir="${test.home}">
                <fileset dir="${test.home}">
                    <include name="**/*Test*.java"/>
                </fileset>
            </batchtest>
        </junit>
    </target>
    <!-- ====== Create a distributable WAR file ====== -->
    <target name="dist"
            depends="unit-test"
            description="Creates deployable WAR file with unit tested code">
       ...
    </target>
```

Continuous Integration

> Continuous Integration is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily — leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. — Martin Fowler

With the popularity of agile development methodologies, *continuous integration* (CI) is fast becoming a common practice in many engineering teams. CI usually includes the following practices:

- Maintaining the source code in a version control system
- Automating the source code build
- Fast builds
- Writing unit test cases for the source code, which are run with every build
- Frequent source code commits to the version control system
- Frequent builds of the "mainline" of the source code on an integration machine

As you can see from this list, you would need additional tools to implement a continuous integration environment; however, automated build systems such as Ant are a cornerstone of all such environments. In the previous section, you saw how an Ant script can be used to automate source code builds, testing, and deployment.

Some popular systems used to implement continuous integration are listed in the following table. All of these tools support Ant scripts.

Tool	Download URL
CruiseControl	`http://cruisecontrol.sourceforge.net`
AntHill	`http://www.urbancode.com/projects/anthill/`
Gump	`http://gump.apache.org`
Continuum	`http://maven.apache.org/continuum/`
LuntBuild	`http://luntbuild.javaforge.com`
BuildBot	`http://buildbot.sourceforge.net`

Further background information on continuous integration practices can be found at `martinfowler.com/articles/continuousIntegration.html`.

Ant Task Reference

A summary of Ant's core tasks is listed in the following table.

Task Name	Description
`ant`	Runs Ant on a build file. This task can be used to build subprojects.
`antcall`	Calls another target within the same build file.
`antstructure`	Generates a Document Type Definition (DTD) for Ant build files.
`apply`	Executes a system command. This task has an optional `os` parameter that specifies the operating system on which the command should be run.
`available`	Sets a property if a resource (for example, file, directory, class, JVM system resource) is available at runtime.
`basename`	Determines the basename of a specified file. Also see `dirname`.
`buildnumber`	This is used to track build numbers.
`bunzip2`	Unzips a file using the BZip2 algorithm.
`bzip2`	Compresses a file using the BZip2 algorithm.
`checksum`	Generates checksum for a file.
`chmod`	Changes permissions of file(s).
`concat`	Concatenates a file or series of files to a file or console.
`condition`	Sets a property if a condition is `true`.
`copy`	Copies a file or set of files to a new location.
`cvs`	Handles CVS modules.
`cvschangelog`	Generates a CVS Change Log in XML format.
`Cvspass`	Adds entries to the CVS `.cvspass` file (same effect as doing a `cvs login`).
`cvsversion`	Retrieves the CVS client/server version.
`cvstagdiff`	Generates a `diff` between two CVS tags (or dates).
`defaultexcludes`	Alters the default excludes for all subsequent processing, or print the current excludes.
`delete`	Deletes a file, a set of files, or a directory.
`dependset`	Manages arbitrary dependencies between files.
`dirname`	Determines the directory path of a specified file.
`ear`	An extension of the `jar` task for handling Enterprise ARchive (EAR) files.
`echo`	Echoes a message to a logger or a listener (the default is to echo to the console).
`exec`	Executes an OS-specific system command.

Task Name	Description
`fail`	Exits the current Ant build.
`filter`	Sets up a token filter. These filters are used by file-copying tasks.
`fixcrlf`	Adjusts a text file for local OS conventions.
`genkey`	Generates a key in a keystore.
`get`	Gets a file from a URL.
`gunzip`	Uncompresses a file using the Gzip protocol.
`gzip`	Compresses a file using the Gzip protocol.
`import`	Imports another build file into the project.
`input`	Prompts for input from the user.
`jar`	Creates a JAR file.
`java`	Executes a Java class within the same virtual machine.
`javac`	Compiles a Java source tree.
`javadoc`	Runs `javadoc` to create project documentation.
`length`	Displays or sets a property containing length information for a string, file, or nested fileset.
`loadfile`	Loads a text file into a property.
`loadproperties`	Loads Ant properties from a file.
`mail`	Sends e-mail.
`macrodef`	Defines a new task using a nested sequential task as template.
`manifest`	Creates a manifest file (used in JAR files).
`mkdir`	Creates a directory.
`move`	Moves a file, a set of files, or a directory to a new location.
`nice`	Sets or queries the priority (`nice` value) of the current thread.
`parallel`	Executes a set of tasks in parallel. Each task executes in its own thread.
`patch`	Applies a "diff" file patch to the original file.
`pathconvert`	Used for converting representations of a path from one form to another.
`presetdef`	Generates a new definition based on an existing definition, with some attributes or elements preset.
`property`	Sets a property.
`record`	Listener to the current build process that records the output to a file.

Table continued on following page

Task Name	Description
replace	Replaces a string with another string in a text file.
rmic	Runs the rmic compiler.
sequential	Specifies a set of tasks to be run in sequence. Typically used for grouping inside a nested parallel task.
signjar	Signs a JAR or ZIP file using the signjar command.
sleep	"Sleep" for a specified amount of time.
sql	Executes an SQL statement via JDBC.
style	Processes a set of documents using XSLT.
subant	Calls a specified target for all defined sub-builds.
tar	Creates a tar archive.
taskdef	Adds a task definition for new (optional) tasks.
tempfile	Sets a property to the name of a temporary file.
touch	Changes the modification time of a file.
tstamp	Sets the timestamp-related properties in the build file.
typedef	Specifies a new type definition for the project.
unjar/untar/unwar/unzip	Extracts a JAR/TAR/WAR or ZIP file.
uptodate	Sets a property if a target file (or set of files) is more current than a source file (or set of files).
waitfor	Blocks until a certain condition is true. Often used in conjunction with the Parallel task.
war	An extension of the JAR task for handling WAR files.
whichresource	Finds a class or resource on the supplied or system CLASSPATH.
xmlproperty	Loads properties from an XML file.
xslt	Processes a set of documents using XML Stylesheet Language Transformations (XSLT).
zip	Creates a ZIP file.

You can find details on the tasks, including all the options they support and examples of their usage, at http://ant.apache.org/manual/coretasklist.html.

Other than these core tasks, Ant can perform a number of optional tasks. More details on these optional tasks can be obtained at http://ant.apache.org/manual/optionaltasklist.html.

Summary

This appendix provided a tutorial introduction to Apache Ant. As you can see from the examples covered here, Ant can be used to construct very elaborate build environments, and to perform tasks that include the following:

- Compiling source code
- Running test cases (when coupled with JUnit)
- Building installable packages
- Deploying applications (for example, using Tomcat's Ant tasks to deploy a Web application)
- E-mailing the status of the test cases or the build to developers

Further information on Apache Ant is available at `http://ant.apache.org/manual/`.

Index

A

access logging valve, 105
implementation, 105
log files, scope, 106–108
AJP connector, 245
Apache JServ protocol, 245
native code Apache modules, 244–245
ports, setting different, 275
AJP protocol (Apache JServ Protocol), 61–62
Ant, 182–183, 597
all target, 606
automated testing, 614
build logs, 610–611
build notifications, e-mail and, 611–613
build.xml file, 604
clean target, 604
command-line options, 601–602
compile target, 185, 187, 605
continuous integration, 615
directory structure, 603
dist target, 605
init target, 604
installation, 598
introduction, 598–601
javadoc target, 605
JSPs, compiling, 608–609
password property, 186
<project> element, 184, 599
scripts
command-line parameters and, 609–610
property files and, 609–610
reusable, 609–610
source control systems and, 613–614
target, 599
task reference, 616–618
Tomcat Manager and, 182–188
undeploying web applications, 188
web applications, building, 602–607
Apache
Apache License, 4
virtual hosting
deployment scenario, 388–389
IP-based, 389–392
name-based, 392–395
Apache Ant. *See* Ant
Apache project, 2
application server configuration, 75–76
Engine component, 76–77
Host component, 79–82
Realm component, 78
APR (Apache Portable Runtime)
connector, 228
attributes, configurable, 229–230
enabling, 229
kernel mode file transfer, 230
OpenSSL support, 231–232
scalable keep-alive poller, 230–231
installation, 47–48
ASF (Apache Software Foundation), 3
authentication
configuration, 358–359
form, 359
form-based, 357
web applications, 355–359

B

BASIC authentication mechanism, 356
bin directory, 52
bootstrap class loader, 207
browsers, Tomcat 6 clustering, 466

C

Catalina engine, 277
CATALINA_HOME, Tomcat startup files, 274–275
catalina.policy, 94–97
catalina.properties, 97
CGI scripts, 13–14
CGI servlet
configuration, 89–90
mappings, 92–93
CGI support, configuration, 232–234
class caching, 210
class loaders
bootstrap, 207
class caching, 210
common class loader, 215–216
custom
creating, 211
reasons for, 211–212
delegation model, 208–209
dynamic class reloading, 217–218
Endorsed Standards Override Mechanism, 209–210, 215
extension, 207–208
lazy loading, 210
namespaces, separate, 210–211
overview, 205–206
packages split among, 218

class loaders (continued)
security
core class restriction, 212–213
delegation, 212
SecurityManager, 213
separate class loader namespaces, 213
singletons, 218–219
system, 208
system class loader, 215
web application class loader, 216–217
XML parsers, 219–220
classes directory, 138
clustering
<Cluster> element, 473–480
fail-over behavioral pattern, 458–459
Farm deployer, 481
JvmRouteBinderValve, 481
listeners, 481
load balancing, 460–461
<Manager> element, 484
master-backup topological pattern, 457–458
performance and, 490–491
Persistent Session Manager
JDBC Store, 487–490
shared file store and, 484–486
response time and, 491
scalability, 456
session sharing, sticky sessions, 461–463
SimpleTcpCluster, 472
<Store> element, 485
Tomcat 6
browsers, 466
configuring, 466–471
cookies, 466
in-memory replication configuration, 472–784
load balancing via Apache mod_jk, 471–472
session management, 465–466
<Valve> element, 480
command-line options, Ant, 601–602
common class loader, 215–216
conf directory, 52–53
configuration
by architecture, Tomcat, 66–67
authentication, tomcat-users.xml file, 86
bootstrapping, 97
CGI servlet, 89–90
context.xml file, default, 82–86
default deployment descriptor, web.xml file, 86–91
invoker servlet, 88
JSPServlet, 88–89
Persistent Session Manager, 115–118
server, 71
server.xml, 72–75
session timeout, 93
SSI servlet, 89–90
Tomcat 6, 70–71
$CATALINA_HOME/conf, 71
web-based GUI configurator, 98–100
tomcat-users.xml file, 86
web application context definitions, 82
web.xml file, 86–91
connection pool managers, c3p0 pooling manager, 326–327
deploying, 327
connector protocols
AJP protocol, 61–62
HTTP protocol, 62–63
connectors
AJP history, 63
AJP JK, 63
JK2, 64
proxy, 64
webapp, 63
context descriptors
server.xml file and, 91–94
web.xml file and, 91–94
Context file, 140
<context-param>, 145
context.xml file, default, 82–86
cookies, Tomcat 6 clustering, 466
c3p0 pooling manager, 326–327
deploying, 327
testing with Tomcat 6 JNDI-compatible lookup, 331

D

DataSource realm, 78
DefaultServlet, 383–384
delegation model, class loaders, 208–209
deployer, 203
deployment descriptor, 140–141
servlet 2.4/2.5-style deployment descriptor, 154–156
context-param, 157
description, 157–158
display-name, 158
distributable, 158
ejb-local-ref, 158
ejb-ref, 158–159
env-entry, 159
error-page, 159–160
filter, 160–161
filter-mapping, 161–162
icon, 162–163
jsp-config, 163–164
listener, 164
locale-encoding-mapping, 165
login-config, 165
message-destination, 165–166, 166
mime-mapping, 166
resource-env-ref, 166
resource-ref, 167
security-constraint, 167–168
security-role, 168
service-ref, 168–169
servlet, 169–170
servlet-mapping, 170
session-config, 170–171
web-app, 156–157
welcome-file-list, 171
servlet 2.3-style deployment descriptor, 141–143
<context-param>, 145
<description>, 144–145

<display-name>, 144
<distributable>, 145
DTD declaration, 143
<env-entry>, 154
<error-page>, 149–150
<filter>, 145–146
<icon>, 144
<listener>, 145–146
<login-config>, 153
<mime-mapping>, 148
<resource-ref>, 151
<security-constraint>, 151–153
<security-role>, 153
<servlet>, 146–147
<session-config>, 147–148
<taglib>, 150–151
<web-app>, 144
<welcome-file-list>, 149
XML header, 143
<description>, 144–145
DIGEST algorithm, 362
DIGEST authentication mechanism, 356
directives, httpd.conf, 277
directories, Tomcat
bin, 52
conf, 52–53
lib, 53
logs, 53
temp, 53
webapps, 53–54
work, 54
<display-name>, 144
<distributable>, 145
downloads, verifying integrity
MD5 DIGEST, 336–338
PGP, 338–340
dynamic class reloading, 217–218
dynamic MBeans, 428

E

Eclipse
Apache Ant and, 593
downloading, 585
remote web application, debugging, 586–588
Sysdeo Tomcat plugin, 589–591
WTP (Web Tools Platform), 591–592
embedded Tomcat, 494
application scenarios, 495
developing with, 495–502
MyWebServer example, 502–503
encryption, SSL and, 377–378
JSSE, 378–379
protecting resources, 381–383
Endorsed Standards Override Mechanism, 215
engine, valves and, 104
Engine component, application server configuration, 76–77
<env-entry>, 154
<error-page>, 149–150
extension class loader, 207–208

F

fail-over behavioral pattern, clustering, 458–459
Farm deployer, 481
file system, securing
Linux, 344–346
Windows, 344–346
<filter>, 145–146
form authenticator valve, 112
form-based authentication, 357
FQDN (Fully Qualified Domain Name), 387–388

G

GNU
GPL (General Public License), 5
LGPL (Lesser General Public License), 5

H

Host component, application server configuration, 79–82
host-manager application, Tomcat, 341, 409
HTTP/1.1 connector, disabling, 276
HTTP Connectors, 222–223
APR connector, 228–232
Comet asynchronous IO support, 228
NIO Connector, 227
Tomcat 6 HTTP/1.1 Connector
configuration, 223–226
configuring Tomcat 6 for SSL, 226–227
HTTP (Hypertext Transfer Protocol), 13
httpd.conf, directives, 277
HTTPS client certificate, 357

I

<icon>, 144
IDEs (Integrated Development Environments)
Eclipse, 585–593
NetBeans, 593–596
IIS (Internet Information Services), 285
Tomcat and, scalable architectures and, 305–307
Tomcat configuration, 286–287
IIS 5 isolation mode, 295–296
installation verification, 287–288
ISAP plug-in as ISS filter, 300–302
ISAP plug-in as web app ext, 302
ISAP plug-in installation, 288–289
JK connector, 288
request forwarding, 291–292
testing, 303
troubleshooting, 303–305
URL rewrite rules, 292
virtual directory creation, 296–299
Windows registry, 292–295
workers, 289–291

in-memory replication configuration, Tomcat 6 clustering, 472–784
in-memory session replication, 463–465
installation
JMeter, 537–538
JVM (Java Virtual Machine)
Linux, 32–34
Windows, 30–32
Tomcat
Linux, 42–44
troubleshooting, 48–49
Windows installer, 36–41
interfaces, Lifecycle, 65–66
invoker servlet, configuration, 88
ISAPI (Internet Services Application Programming Interface), 285
plug-in, 285–286

J

JAAS realm, 78
configuration, 374–377
Java
APIs, 6–7
JCP (Java Community Process), 7
JSP (JavaServer pages), 19–20
Java EE, 6
APIs, 7–8
application servers, 8
JavaServer Pages. *See* JSP (JavaServer pages)
jconsole, 447–450
JCP (Java Community Process), 7
JDBC (Java Database Connectivity), 309
alternative configuration, 326
connections
JNDI mapping, 330–331
obtaining without JNDI lookup, 327–329
database connection pooling, 313–315
DataSource, configuration, 124–126
drivers, types, 312–313
overview, 310–311
connections, establishing and terminating, 311
versions, 311–312
realms, 363–368
digested passwords, 365
MySQL-based, 367–368
MySQL tables, 365–366
testing, 368
JDBC realm, 78
JMeter
alternatives to, 558
assertions, 551–552
config elements, 550
HTTP authorization manager, 550
HTTP cookie manager, 550–551
HTTP header manager, 550
HTTP request defaults, 551
distributed load testing, 554–555
HTTP Proxy Server, 552–554
installation, 537–538
interpreting test results, 555–557
listener, 544
assertion results, 546
data listeners, 545–546
visualization listeners, 545
logic controller, 547
if controller, 548
interleave controller, 547–548
loop controller, 548
module controller, 548
once only controller, 548
random controller, 549
recording controller, 549
simple controller, 548
switch controller, 548
throughput controller, 549
while controller, 548
sampler, 549–550
test plans, 538–542
timer, 543–544
JMX (Java Management Extension), 419, 422
agent level, 425
agent services, 427
connectors, 426–427
MBean Server, 425
protocol adapters, 426–427
architecture, 422–423
distributed services level, 427–428
instrumentation level, 424
manageable nested components, 433–434
Manager, 434
Realm, 433
Valve, 434
manageable resource object, 436–441
Cache, 440–441
Environment, 436
exposed internal Tomcat objects, 439
NamingResources, 436
RequestProcessor, 440
Resource, 436–437
ResourceLink, 437
Servlet, 439
ThreadPool, 441
WebModule, 437–439
manageable runtime data objects, 435–436
Role, 435–436
User, 435
UserDatabase, 435
manageable Tomcat 6 architectural components
Connector, 431–433
Engine, 431
Server, 430
Service, 430
Manager proxy and, 441–452
remote API, 428
JNDI (Java Naming and Directory Interface)
emulation, 315–316
JDBC DataSource, configuration, 124–126
mail sessions, configuration, 126–128
overview, 118–119
pooling, 315–316

realms, 369–370
configuring, 370, 373
installing LDAP driver, 370
LDAP schema, 371
populating directory, 371–373
roles, removing, 373
users, removing, 373
resource configuration, 118, 121
JDBC resource, 321–324
MySQL test database, 319–321
read-only user, 321
<Resource> element, 122, 317–319
<ResourceLink> element, 123
<ResourceParams> element, 123
testing, 324–325
resources, 120
Tomcat and, 119–120
JNDI realm, 78
JSP (JavaServer pages), 19–20
EL (Expression Language), 23
Model 1 architecture, 20–21
Model 2 architecture, 21–22
MVC architecture, 24–25
tag libraries, 22–23
JSPServlet, configuration, 88–89
JSSE (Java Secure Socket Implementation), certificate keystore, 378–379
JULI
filters, 529
formatter, 529
handler, 528
levels, 528
log files, 531–532
logger, 528
tutorial, 529–531
JVM (Java Virtual Machine)
installation, 29–30
Linux, 32–34
Windows, 30–32
Security Manager
enabling system, 350
grant entry, 347–348
permissions, 347, 348–350
recommended practices, 353–355
Tomcat and, 350–353
jvmRoute, 276
JvmRouteBinderValve, 481

L

lazy loading, class loaders, 210
LGPL (Lesser General Public License), 5
lib directory, 53, 139
Lifecycle interface, 65–66
Lifecycle listeners, configuration
APR Lifecycle listener, 132
default, removing, 131–132
events sent by Tomcat components, 129
<Listener> element, 129–130
MBeans, 130–131
native SSL engine configuration, 132–133
lifecycleEvent() method, 66
LifecycleListener interface, 65–66
Linux
file system, securing, 344–346
JVM (Java Virtual Machine), installation, 32–34
Tomcat, installation, 42–44
<listener>, 146
<Listener> element, 129–130
load balancing
clustering, 460–461
testing load balancer, 279–280
different load factors, 283–284
round-robin behavior, 281–283
sticky sessions, 280–281
Tomcat
Apache and, 273–279
Apache mod_jk, 471–472
log files, scope, 106–108
logging
log4j
architecture, 506–509
configuration, 509–514
custom formatting of messages, 520–521
e-mail log messages, 522–523
installation, 509–514
log files, 526–527
logging from web application, 515–516
logging messages as HTML, 521–522
logging to console, 517
logging to file, 517
logging to multiple destinations, 517–518
nested diagnostic context, 525–526
NT event log, 523–525
rolling log files by date, 518–519
rolling log files by size, 518
separating messages by level, 519–520
specific packages or classes, 520
tutorial, 514–515
Tomcat 5 changes, 505–506
<login-config>, 153
log4j
appender, 507–508
architecture, 506
configuration, 510
programmatic, 511–512
XML configuration file, 512–513
custom formatting of messages, 520–521
e-mail log messages, 522–523
filters, 509
layout, 509
levels, 508
log files, 526–527
logger, 507
logging
to console, 517
to file, 517
messages as HTML, 521–522
to multiple destinations, 517–518
from web application, 515–516

log4j (continued)
nested diagnostic context, 525–526
NT event log, 523–525
properties file, 510–511
rolling log files
by date, 518–519
by size, 518
separating messages by level, 519–520
specific packges or classes, 520
tutorial, 514–515
logs directory, 53

M

mail sessions, configuration, 126–128
manager application, 341
access, enabling, 176–178
configuration
context entry, 178
deployment descriptor, 178–179
Manifest file, 139–140
master-backup topological pattern, clustering, 457–458
MBeans, 428
attributes, modifying, 444–446
dynamic, 428
model, 429
open, 429
standard, 428
McCool, Rob, 2
MD5 DIGEST, verifying, 336–338
Memory realm, 78
META-INF directory
Context file, 140
Manifest file, 139–140
methods, lifecycleEvent(), 66
mime mappings, 93–94
<mime-mappings>, 148
model MBeans, 429
mod_jk module, 253–259
mod_proxy module, 259–263
MySQL, Tomcat users, 366–367

N

namespaces, separate, 210–211
NCSA (National Center for Supercomputer Applications), 2
NetBeans
remote web applications, debugging, 594–595
web applications, debugging, 596

O

open MBeans, 429

P

performance testing
JMeter
installation, 537–538
running, 537–538
measuring performance, 535–537
scalability, 534–535
user's perspective, 535
what to measure, 533–534
performance tuning
baseline, 563
best practices, 561–564
bottlenecks, 564
configuration attributes, 239–240
diagnosing performance issues, 564–566
JSPs, precompiling, 569–571
JVM parameters, 567–569
TCP/IP stack tuning, 240–241
test bed setup, 562–563
Tomcat configuration, tuning, 571–582
web servers, static content, 582–583
Persistent Session Manager
configuration, 115–118
<Manager> element, 116–117
persistent sessions, need for, 115
PGP, 338–340
ports
AJP connector, setting different, 275–276
server, setting different, 275–276
proxy server, 238–239

R

RDBMSs (relational database management systems), 309
Realm component, application server configuration, 78
realms
JAAS, configuration, 374–377
JDBC, 363–368
JNDI, 369–370
configuring, 370, 373
installing LDAP driver, 370
LDAP schema, 371
populating directory, 371–373
roles, removing, 373
users, removing, 373
security
roles, 360
users, 360
UserDatabase, 361–363
web applications, 355–359
reference implementation (RI), 3
remote monitoring, 450–452
replication, <Valve> element, 480
request dump valve, 105, 114–115
request filtering valve, 105
configuration, 113–114

remote address filter, 112–113
remote host filter, 113
<resource-ref>, 151
RI (reference implementation), 3
roles, security realms, 360
ROOT application, 341

S

scalability
clustering and, 456
performance testing and, 534–535
security
class loaders
core class restriction, 212–213
delegation, 212
separate class loader namespaces, 213
SeparateManager, 213
DefaultServlet, 383–384
downloads, verifying integrity, 336–340
JVM (Java Virtual Machine), 346–347
Security Manager, 347–350
Security Manager, recommended practices, 353–355
Security Manager, Tomcat and, 350–353
Linux file system, 344–346
realms
JDBC, 363–368
roles, 360
UserDatabase, 361–363
users, 360
server installation, 340–342
SSL, encryption with, 377–383
virtual hosting, 409–410
web applications
authentication, 355–359
realms, 355–359
Windows file system, 344–346
<security-constraint>, 151–153
<security-role>, 153
servers
configuration, 71
server.xml, 72–75
installation, securing, 340–342
ports, setting different, 275–276
proxy server, 238–239
server.xml
Connector component, 74–75
context descriptors and, 91–94
Server component, 72–73
Service component, 73–74
web.xml and, 91–94
<servlet>, 146–147
servlet container, 1
servlets (Java)
accessing, 18–19
interface, 15–17
mappings, 90–91
<session-config>, 147–148
session management, Tomcat 6, clustering, 465–466
session sharing, clustering
in-memory session replication, 463–465
sticky sessions
no clustered session sharing, 461–462
Persistence Manager and JDBC-based store, 463
Persistence Manager and shared file store, 462–463
session timeout, configuration, 93
SimpleTcpCluster, 472
single sign-on valve, 105
configuration, 111–112
implementation, 108–112
multiple sign-on without, 109–111
SLL (Secure Sockets Layer), encryption with, 377–378
JSSE, 378–379
special account, 342–344
SSI servlet
configuration, 89–90
mappings, 92–93
SSI support, configuration, 234–238
SSL (Secure Sockets Layer), 305
configuration
mod_ssl for Apache, 264–269
SSL-enabled Apache-Tomcat setup, 271–273
testing SSL-enabled Apache setup, 269–271
encryption, protecting resources, 381–383
standard MBeans, 428
sticky sessions, load balancing, 460–461
system class loader, 208, 215

T

tag libraries (JSP), 22–23
<taglib>, 150–151
tags directory, 138
temp directory, 53
Tomcat
Apache License, 4
application servers, 9
architecture, 54–55
classes, 59
configuration by, 66–67
Connectors, 56, 59–64
Context, 58–59
Engine, 56–57
Host, 58
lifecycle, 64–66
Loggers, 58
Realm, 57
Server, 55
Service, 56
Valves, 57–58
building from source
building source release, 45
downloading source release, 44
subversion repository, 45

Tomcat (continued)
Connector architecture, 59–60
connector protocols, 61–63
connectors, choosing, 63–64
deployer, 203
directories, 51
bin, 52
conf, 52–53
lib, 53
logs, 53
temp, 53
webapps, 53–54
work, 53–54
distributing, 4
embedded, 494
application scenarios, 495
developing with, 495–502
MyWebServer example, 502–503
host-manager application, 409
installation
distribution, deciding on, 34–35
downloaded file, verifying, 35–36
Linux, 42–44
Windows, ZIP file, 41–42
Windows installer, 36–41
installation directory, 46–47
load balancing, Apache and, 273–279
running with special account, 342–344
server, installation, securing, 340–342
virtual hosting, 395–396
Apache, configuring, 406–408
with Apache, 405–406
deployment scenario, 396–398
Java Security Manager restrictions, 416–417
JVMs, memory limits, 414–416
JVMs, separate, 410–414
Tomcat as standalone server, 398–405
web servers and, 9–10
workers, multiple, 246–251
Tomcat 6
clustering
browsers, 466
configuring, 466–471
cookies, 466
in-memory replication configuration, 472–784
load balancing via Apache mod_jk, 471–472
session management, 465–466
configuration, 70–71
$CATALINA_HOME/conf, 71
connecting Tomcat with Apache
apache web server configuration, 252–253
configuration, 251–252
mod_jk module, 253–259
mod_proxy module, 259–263
web-based GUI configurator, 98–100
tomcat-docs application, 341
Tomcat Manager
Ant and, 182–188
undeploying web applications, 188
errors, 200–201
HTTP requests, 189–190
deploying applications, Tomcat 6, 191–192
deploying applications from local path, 192–194
deploying applications remotely, 192
deploying new applications, 190–191
displaying session statistics, 198–199
installing applications, Tomcat 6, 191–192
listing deployed applications, 190
listing JNDI resources, 195–196
listing OS and JVM properties, 196
querying Tomcat internals, JMX Proxy servlet and, 199–200
reloading applications, 194–195
setting Tomcat internals, JMX proxy servlet and, 200
starting stopped applications, 197
stopping applications, 196–197
undeploying web applications, 198
Tomcat project, 3
tomcat user, 343–344
tomcat-users.xml file, 86
troubleshooting, Tomcat installation, 48–49

U

user, non-privileged, 343
UserDatabase realm, 78, 361–363
users, security realms, 360
user's perspective, performance and, 535

V

<Valve> element, 480
valves, 104–105
access logging, 105
implementation, 105
log files, scope, 106–108
Engine and, 104
form authenticator, 112
request dump, 105, 114–115
request filtering, 105
configuration, 113–114
remote address, 112–113
remote host, 113
single sign-on, 105
configuration, 109–111
implementation, 108–112
multiple sign-on without, 109–111
virtual hosting
Apache
deployment scenario, 388–389
IP-based, 389–392
name-based, 392–395
host-manager application, Tomcat, 409
performance, 409–410
security, 409–410
stability, 409–410
Tomcat
Apache, configuring, 406–408
with Apache, 405–406
deployment scenario, 396–398

Java Security Manager restrictions, 416–417
JVMs, memory limits, 414–416
JVMs, separate, 410–414
as standalone server, 398–405

W

<web-app>, 144
web application class loader, 216–217
web applications
authentication, 355–359
building, 26–27
CGI scripts and, 13–14
contents, 135–136
context definitions, 82
deploying, 182
distributing, 26–27
HTTP, 13
managing, 181–182
META-INF directory
Context file, 140
Manifest file, 139–140
public resources, 136–137
realms, 355–359
security, 201–203
servlets (Java), 14–15
accessing, 18–19
interface, 15–17
technologies, 25–26
undeploying, failure during, 188
WEB-INF directory
classes directory, 138
lib directory, 139
tags directory, 138–139
WEB-INF directory
classes directory, 138
lib directory, 139
tags directory, 138–139
Web interface
server status, displaying, 180–181
web applications
deploying, 182
managing, 181–182
web server, front ending with, 241–242
webapps directory, 53–54
web.xml file, 86–91
context descriptors and, 91–94
server.xml and, 91–94
welcome file, 94
<welcome-file-list>, 149
Windows
file system, securing, 344–346
JVM (Java Virtual Machine), installation, 30–32
Windows installer, Tomcat
default installation, viewing, 40
environment variables, 37
port numbers, assigning, 40–41
Service component, 36–37
testing installation, 37–40
work directory, 54
workers.properties, configuration, 277–279

Programmer to Programmer™

BROWSE BOOKS | P2P FORUM | FREE NEWSLETTER | ABOUT WROX

Get more Wrox at Wrox.com!

Special Deals

Take advantage of special offers every month

Free Chapter Excerpts

Be the first to preview chapters from the latest Wrox publications

Unlimited Access. . .

. . . to over 70 of our books in the Wrox Reference Library. (see more details on-line)

Forums, Forums, Forums

Take an active role in online discussions with fellow programmers

Meet Wrox Authors!

Read running commentaries from authors on their programming experiences and whatever else they want to talk about

Browse Books

.NET
SQL Server
Java
XML
Visual Basic
C#/C++

Join the community!

Sign-up for our free monthly newsletter at
newsletter.wrox.com

powered by books24x7

Programmer to Programmer™

Take your library wherever you go.

Now you can access more than 70 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the **Wrox Reference Library.** For answers when and where you need them, go to wrox.books24x7.com and subscribe today!

Find books on

- ASP.NET
- C#/C++
- Database
- General
- Java
- Mac
- Microsoft Office
- .NET
- Open Source
- PHP/MySQL
- SQL Server
- Visual Basic
- Web
- XML

wrox™

www.wrox.com